Blockchain Ethereum

Fundamentos de arquitetura, desenvolvimento de contratos e aplicações

Casa do Código | alura

JOÃO KUNTZ

Sumário

- ISBN
- Sobre o autor
- Sobre o livro
- Parte 1 - BLOCKCHAIN COMO TECNOLOGIA
  - 1 Introdução
  - 2 Blockchain para principiantes
  - 3 A Blockchain Ethereum
  - 4 Anatomia das aplicações Ethereum
- Parte 2 - DESENVOLVENDO PARA A BLOCKCHAIN ETHEREUM
  - 5 Preparando o ambiente de desenvolvimento
  - 6 Linguagem de programação Solidity
  - 7 Implementando Smart Contracts e Tokens
- Parte 3 - SEGURANÇA E ARMAZENAMENTO NA BLOCKCHAIN ETHEREUM
  - 8 Segurança na Rede Ethereum
  - 9 Armazenamento e analytics na Rede Ethereum
  - 10 Referências

ISBN

Impresso: 978-85-5519-294-4

Digital: 978-85-5519-293-7

Caso você deseje submeter alguma errata ou sugestão, acesse http://erratas.casadocodigo.com.br.

Sobre o autor

João Kuntz

Tecnólogo em análise e desenvolvimento de sistemas pela Faculdade de Tecnologia da Baixada Santista Rubens Lara (FATEC-RL), especialista em Engenharia da Qualidade de Software pelo Centro Universitário Senac, possui MBA em Arquitetura de Software pelo Instituto de Gestão e Tecnologia (IGTI) e é Mestre em Engenharia Mecânica (controle e automação mecânica), com pesquisas na aplicação de Redes Neurais Artificiais Paraconsistentes, pela Universidade Santa Cecília.

Foi palestrante na TDC Future 2021 na trilha de blockchain, falando sobre a criação de *smart contracts* em projetos DeFi, e também é professor no MBA de Desenvolvimento Descentralizado e Distribuído da FIAP nas disciplinas de Arquitetura de Blockchain de Rede Pública e de Desenvolvimento de Smart Contracts & Tokenização.

Antes disto, formou-se bacharel em química e trabalhou por alguns anos com análises espectroscópicas até descobrir e se encantar pela informática.

Atua como Arquiteto de Software em projetos internacionais, com uma marcante passagem profissional pelo México, tendo liderado implementações simultâneas na Guatemala, El Salvador, Honduras e Colômbia, com ênfase na arquitetura de integrações com sistemas SAP e SalesForce e no desenho técnico de sistemas com arquitetura de microsserviços, além de ter sido responsável pela criação de estratégias de CI/CD e implementação de práticas DevOps e SRE com os times.

Apaixonado pela cultura *data-driven*, pela arquitetura de dados e a computação em nuvem, possui certificações Microsoft e AWS.

Grande entusiasta da tecnologia de blockchain, acredita que as empresas que queiram garantir suas vantagens competitivas frente a Web 3.0 devem começar a pensar desde já em suas estratégias em aplicações descentralizadas.

Sobre o livro

Para quem é este livro

Este livro foi planejado com vistas a suprir uma lacuna existente na literatura técnica, em português, sobre a tecnologia blockchain, especificamente sobre o Ethereum – uma blockchain pública, capaz de hospedar e executar trechos de código, sendo, portanto, classificada como uma blockchain de uso geral.

Nossa preocupação aqui foi a de apresentar os fundamentos do funcionamento desta tecnologia, o que veio como uma resposta ao desafio de manter um livro técnico atualizado, considerando a velocidade com que vemos atualizações de produtos e componentes.

Se, por um lado, a tecnologia como um todo evolui rapidamente, os fundamentos são mantidos ao longo dos anos – sendo seu conhecimento necessário e pertinente para profissionais de diferentes níveis de senioridade, em diferentes momentos, o que justifica nossa opção.

Nosso público-alvo abrange uma grande gama de pessoas interessadas no assunto. Justamente por não ser um livro baseado inteiramente em um *framework* muito específico de trabalho, ele pode ser aproveitado com a mesma intensidade por pessoas leigas, interessadas em descobrir como uma blockchain funciona pelo simples fato de este ser um assunto bastante comentado ultimamente, como por analistas de negócio e profissionais em posição estratégica, que queiram entender como ela pode trazer ganhos de competitividade às suas organizações, e, da mesma maneira, por desenvolvedores e pessoal estritamente técnico, que queiram descobrir como é possível escrever aplicações que se baseiem em uma blockchain para funcionar, aumentando seu portfólio de tecnologias de trabalho.

Conhecimentos necessários

Por ser um livro de fundamentos, não existem pré-requisitos sobre o assunto para sua leitura. Iniciamos nossa jornada explicando cada conceito necessário a partir do zero, de forma que todos poderão aproveitar as informações da mesma maneira.

Por outro lado, como tratamos aqui de sistemas computacionais, esperamos que você tenha um mínimo de um conhecimento sobre informática, e tenha algumas noções sobre lógica de programação. Por exemplo, em algum momento citamos a existência de compiladores, mas não explicamos o que são e para que servem – o que é um conhecimento que esperamos que o leitor ou leitora tenha, ou seja capaz de buscar por meios próprios.

Mais para o final do livro temos alguns capítulos dedicados à codificação de *smart contracts*; neles, montamos nosso ambiente utilizando o Node.js, com fins meramente ilustrativos.

Apesar de não ser necessário o conhecimento sobre este *runtime*, já que cada passo terá sido detalhado – podendo ser simulado mesmo por pessoas que não o conheçam – não entramos em detalhes específicos sobre ele ou seu funcionamento, por exemplo, quando citamos o uso de comandos do *Node Package Manager (NPM)* sem dar detalhes sobre sua execução ou sobre a estrutura de seus pacotes. Apesar de não impedir o entendimento do assunto, caso você tenha dificuldades em momentos como estes, sugerimos a busca por informações complementares antes de prosseguir para os próximos capítulos.

O que esperar deste livro

Um dos maiores desafios que encontramos quando nos propomos a escrever um livro técnico, especialmente na área da informática, é o

de que a informação contida nele vira obsoleta rapidamente.

Tendo isso em mente e, indo de acordo com os grandes autores da Ciência da Computação, aqui teremos como foco os fundamentos da blockchain e sua programação, usando como ferramentas as principais tecnologias aplicadas, sem dar ênfase aos projetos paralelos nem frameworks e aplicações muito específicos. Como os fundamentos não se alteram ao longo do tempo (pelo menos não com a mesma velocidade que a tecnologia), mantemos, assim, uma obra atual e interessante para o público ao longo dos anos.

A documentação oficial do projeto seguramente dará à leitora e ao leitor todas as respostas sobre novidades e procedimentos, e consultando-a, temos a certeza de estarmos sempre com a informação mais atualizada sobre a versão de trabalho. O link para o projeto Ethereum é o https://ethereum.org/, e também segue citado na bibliografia.

Esta não é, de forma alguma, uma publicação sobre criptomoedas (ainda que elas sejam mencionadas eventualmente). Não serão dados aqui conselhos sobre investimentos, tampouco opiniões sobre valor de carteiras, transações, nem mesmo sobre plataformas de ativos digitais. Existem diversos canais na Internet nos quais o leitor pode se informar, se este for o interesse.

Este também não é um livro com foco no ensino de programação. Introduziremos aqui uma linguagem de programação de alto nível — o *Solidity* — e demonstraremos como ela pode ser aplicada na criação de *Smart Contracts* — trechos de código executados na blockchain. Assume-se que a leitora ou o leitor já possua certa familiarização com algoritmos, lógica de programação e que conheça conceitos básicos de programação orientada a objetos. Assim, deixaremos de abordar temas básicos, como o funcionamento da avaliação de condicionais com *if/else*, por exemplo, por entendermos ser este um assunto de conhecimento geral.

No capítulo 1 introduziremos os conceitos gerais sobre a tecnologia por trás da cadeia de blocos e o que ela se propõe a resolver. Falaremos sobre a Web 3.0, a evolução da blockchain ao longo da história e sua adoção, citaremos alguns serviços de plataformas com soluções para diferentes blockchains, até chegarmos no Ethereum.

A partir do capítulo 2 veremos como é o funcionamento da blockchain a partir de um ponto de vista técnico. Serão elucidados temas como criptografia, algoritmos de consenso, mineração, a importância do controle do tempo no processo de mineração, entenderemos qual a finalidade dos quebra-cabeças lógicos, veremos o que são transações e, ainda, entenderemos termos como *Proof-of-Work* (PoW), *Proof-of-Stake* (PoS), *Proof-of Authority* (PoA), *Mempool*, *Nonce*, blocos órfãos, entre outros. Isso nos permitirá entender como a blockchain é uma tecnologia tão segura, como é garantida a imutabilidade das transações e ainda, como se garante a ininterrupção dos seus serviços.

Em seguida, no capítulo 3, nossa atenção se virará totalmente para a Rede Ethereum. Serão apresentados seus principais conceitos e filosofias, entenderemos como são processadas as transações dentro dela, discutiremos máquinas de estado, blocos do Ethereum, falaremos sobre os *forks* do projeto — que modificaram a cadeia original —, trataremos de temas como contas e carteiras, falaremos sobre o *Ethereum Gas*, definiremos mensagens, comprovantes de transação, dificuldade dos blocos, contratos e apresentaremos, com detalhes, a EVM. Terminaremos o capítulo falando do Ethereum 2.0, uma atualização da rede que trará vantagens competitivas extremamente importantes, e repassaremos sua história.

A partir do capítulo 4 abandonamos os conceitos estritamente teóricos passamos a discutir os principais tipos de aplicações e tecnologias com as quais trabalharemos ao implementarmos nossas soluções no Ethereum. Falaremos sobre os contratos inteligentes, criação de aplicações descentralizadas, utilização de mecanismos de DeFi e NFTs e apresentaremos um desafio importante para o

Ethereum: sua capacidade de escalabilidade. Veremos como esse tema vem sendo tratado, e as alternativas disponíveis.

No capítulo 5 começaremos a preparar nosso ambiente para o desenvolvimento de *smart contracts*. Conheceremos alguns componentes que serão utilizados e faremos suas instalações, passo a passo, de forma que todos estejamos com um ambiente pronto para o uso nos capítulos posteriores.

No capítulo 6, finalmente, estudaremos o *Solidity*, a principal linguagem de programação utilizada para o desenvolvimento de contratos publicados no Ethereum. Ao final, além da sintaxe de uma nova linguagem de programação, você conhecerá detalhes de sua arquitetura como tipos de dados, declarações de armazenamento de variáveis, funções globais, geração de eventos, criação de funções, entre outros.

Todo o conhecimento adquirido será colocado em prática no capítulo 7. Aqui, desenvolveremos dois *smart contracts* diferentes, e trabalharemos com a "tokenização", isto é, a criação de *tokens* fungíveis e não fungíveis (NFTs) para o Ethereum. Veremos passo a passo como as ferramentas disponibilizadas pelo *Solidity* podem ser utilizadas e organizadas, de forma a garantir a execução de trechos de código de forma eficiente, com menores custos de transação. Também testaremos a eficácia de nosso código a partir da escrita de testes unitários, em que veremos o resultado de nossas execuções, em tempo real.

No capítulo 8 discutiremos aspectos de segurança da Rede Ethereum. Veremos os principais tipos de ataque para os quais uma rede blockchain é vulnerável e também listaremos algumas das principais vulnerabilidades que devem ser tratadas no desenvolvimento de nossos códigos, durante a criação dos nossos *smart contracts*.

Finalmente, no capítulo 9, falaremos sobre o armazenamento de arquivos em uma blockchain e finalizaremos tratando sobre um

assunto extremamente atual: a consulta e o consumo de dados de uma blockchain com vistas à geração de *insights*, utilizados por cientistas de dados e consumidos por aplicações de inteligência de mercado. Finalizaremos apresentando o projeto *The Graph* ("O Grafo") que, através de um protocolo de indexação próprio, permite a realização de consultas do tipo *GraphQL* no Ethereum, servindo como uma importante ferramenta para o profissional que queira trabalhar com *analytics* no contexto da blockchain.

Parte 1 - BLOCKCHAIN COMO TECNOLOGIA

Capítulo 1
Introdução

1.1 Web 3.0 e a descentralização

Falar sobre a Web 3.0 não é algo original; não apenas as suas definições vêm sendo exaustivamente revisitadas em publicações técnicas e acadêmicas recentes, como também temos sido testemunhas de sua evolução e — cada vez mais — vamos nos acostumando a interagir com ela em nosso cotidiano. Para quem não está familiarizado com o termo, ele traduz (de forma bastante generalista) o movimento de adoção das tecnologias de Inteligência Artificial (IA) às tarefas banais, do dia a dia, de forma a criar e oferecer conteúdos e experiências adaptados às necessidades do usuário, de forma otimizada.

Se por um lado este movimento é criador de oportunidades, sendo complementado pelas evoluções do 5G e da Internet das Coisas (IoT), por exemplo, ele traz como grande revés as questões do armazenamento e compartilhamento de dados — importantes fatores a serem considerados quando tratamos de questões de segurança. Afinal, o quanto nos sentimos confortáveis em compartilhar com as organizações nossos dados sobre interesses, aquisições recentes, históricos de busca e afins?

No Brasil, a Lei N° 13.709 de 14 de agosto de 2018 — Lei Geral de Proteção de Dados Pessoais (LGPD) — se propõe a “proteger os direitos fundamentais de liberdade e de privacidade e o livre desenvolvimento da personalidade da pessoa natural”, conforme enunciado do próprio Ministério da Defesa, garantindo direitos

fundamentais como a privacidade e a intimidade, inclusive nos meios digitais. Ao redor do mundo, regulamentações como o Regulamento Geral sobre a Proteção de Dados (GDPR), na União Europeia, e a Lei de Privacidade do Consumidor da Califórnia (CCPA), nos Estados Unidos, por exemplo, também vêm versado sobre o tema, demonstrando a relevância que o assunto tomou nos últimos anos.

Corolário disto, a “descentralização” tornou-se um pilar desta nova Web (também conhecida como *Web Semântica* ou mesmo "Web Inteligente"). Dentro deste novo contexto, deixamos de compartilhar com as organizações a maioria dos nossos dados (tente imaginar por um momento a quantidade de informação que as empresas já construíram sobre você, analisando apenas seu comportamento nas redes sociais), passando a adotar um papel mais ativo: através de tecnologias específicas (como a criptografia) paramos de deixar pegadas digitais, utilizadas por grandes empresas para construir informações sobre nós, e passamos a tomar uma postura ativa frente a isso, sendo nós mesmos os responsáveis por moldar, com o auxílio da IA, as nossas próprias experiências, com maior liberdade.

Em artigo publicado no Medium em 2018 e com livre acesso, Matteo Zago apresenta de forma brilhante uma comparação entre a Web 1.0 até os dias de hoje e cita algumas vantagens esperadas da Web 3.0 e sua descentralização, como:

1. A ausência de um ponto central de controle — empresas ou governos deixam de controlar dados pessoais e perdem o poder de censurar e bloquear o acesso a serviços;

2. A propriedade dos dados pessoais passa a ser do próprio usuário, protegendo a privacidade de uso;

3. Redução no número de ataques de *hackers* e vazamento de dados, uma vez que a informação é descentralizada e que um ataque *hacker* efetivo necessitaria conseguir acessar todos os computadores da rede ao mesmo tempo para ter efeito, ao

passo que a segurança dos dados é garantida pela criptografia e pelo controle pessoal. Como os dados não são armazenados em um único repositório, eles não podem ser acessados, vazados, capturados ou vendidos;

4. Interoperabilidade, já que as aplicações deixam de rodar em sistemas operacionais de determinados sistemas, e ficam disponíveis na rede;

5. O acesso aos serviços não pode ser interrompido nem bloqueado, mesmo em questões geopolíticas em que a proibição de acesso é dada por governos censores.

Apesar de a descentralização parecer ser algo trivial quando mencionada, ela traz consigo uma série de desafios de implementação e manutenção, que por si só são capazes de eliminar várias tecnologias candidatas a serem utilizadas como base para o desenvolvimento de aplicações. Falamos aqui de arquiteturas de aplicação em que, idealmente, não sejam permitidas a interceptação e leitura de dados de identificação.

Arquiteturas *serverless* ("sem servidores") seriam boas candidatas, desde que tampouco estivessem contidas em uma rede ou repositórios (afinal, não podemos armazenar os dados dos nossos usuários). Por mais seguras que sejam suas transações, no entanto, elas também possuem suas limitações. É justamente cobrindo este hiato que a blockchain aparece como uma alternativa extremamente promissora para o uso na nova Web.

1.2 Evolução da tecnologia blockchain

Idealizada no início da década de 1990, a Cadeia de Blocos (tradução aproximada para o termo blockchain) é uma tecnologia, erroneamente, quase que exclusivamente associada ao mercado de

criptomoedas, porém, com grande potencial de uso na Web 3.0 — justamente por sua natureza: descentralizada e segura.

Ainda que seja inegável a importância do Bitcoin para o desenvolvimento da tecnologia blockchain, por exemplo, restringir seu potencial de aplicação apenas a isso chega a ser injusto!

Os aspectos técnicos da blockchain serão discutidos com maior aprofundamento no próximo capítulo, contudo é mister dedicar aqui alguns parágrafos sobre sua origem.

As primeiras ideias do seu desenvolvimento surgem a partir de uma publicação de apenas cerca de dez páginas intitulada *How to Time-Stamp a Digital Document* ("Como marcar a data e hora em um documento digital", em tradução livre), por Stuart Haber e W. Scott Stornetta, em 1991. Nela, os autores discutiram a necessidade de criar mecanismos para assegurar que documentos digitais (que então começavam a aparecer) fossem armazenados de forma segura, tal que houvesse garantia de que não fossem ser modificados aleatoriamente ou de forma proposital (como em uma ação para a realização de fraudes).

Aqui os autores sugeriram, entre outras coisas, a utilização de mecanismos de *hashing*, com fins de garantir a integridade dos dados; assinaturas digitais, garantindo a origem dos dados; e a distribuição de seu conteúdo, tornando inexistente a figura do repositório único, este, capaz de ser acessado diretamente e manipulado, ao que eles chamaram de “Confiança distribuída” (*distributed trust*, também conhecida pela expressão *trustless*).

No ano seguinte os mesmos autores sugeriram a adoção das chamadas Árvores de Merkle em seu modelo sobre o controle de documentos digitais como forma de aumentar a eficiência do sistema proposto, permitindo a inclusão de um número maior de dados dentro de um único bloco de transações. Foram necessários outros 16 anos, no entanto, até que a próxima grande evolução aparecesse na história da blockchain.

Em 2008, Satoshi Nakamoto, pseudônimo utilizado por uma pessoa (ou possivelmente um grupo de pessoas), apresentou a idealização da blockchain tal como utilizado hoje, em uma publicação intitulada *Bitcoin: A Peer-to-Peer Electronic Cash System* ("Bitcoin: Um sistema ponto a ponto de dinheiro eletrônico"). Nele, o autor descreveu o funcionamento de um mecanismo de pagamento independente de instituições financeiras, centralizadoras de informação, considerando os aspectos críticos de segurança que este tipo de operação demanda. Após a publicação, Nakamoto desapareceu e, até hoje, nada se sabe sobre sua identidade.

Uma rede blockchain é, efetivamente, uma rede distribuída ponto a ponto, segura, baseada no funcionamento dos Livros Razão da contabilidade, usada para manter registros de transações através de múltiplos computadores, sem a necessidade da figura da autoridade central arbitral.

Ela é constituída por blocos que contêm informações sobre transações, tendo diferentes mecanismos de manutenção da integridade dos dados como a utilização de mecanismos de *hashing* que envolvem tanto o bloco anterior como o bloco presente (tornando-se inválida quando qualquer alteração é feita em um bloco), mecanismos de assinatura via o uso de chaves públicas e privadas, controles de data/hora de criação de blocos, entre outros.

Por residir em uma estrutura distribuída, *peer-to-peer* (P2P, "ponto a ponto"), seus dados são naturalmente distribuídos ao longo de diferentes computadores participantes da rede, existindo, assim, um grande número de cópias dos blocos, o que serve, ainda, como mecanismo de controle.

1.3 Blockchain: percepção e adoção

Uma pesquisa de 2020 publicada pela Delloite (disponível nas referências bibliográficas) sobre a percepção da adoção da blockchain considerando executivos de 14 países (incluindo o Brasil, com 50 entrevistados) mostra que, para 55% dos respondentes, a blockchain é vista como uma das tecnologias a serem usadas dentro das 5 principais prioridades estratégicas das empresas dentro dos próximos 24 meses, sendo que 88% estão confiantes de que ela é uma tecnologia que eventualmente será dominante no mercado, e que 83% afirmaram que suas organizações perderão vantagens competitivas se não a adotarem como tecnologia.

Na mesma pesquisa, 39% dos entrevistados afirmaram já possuírem alguma implementação de blockchain em seu ambiente de produção em 2020, um aumento significativo quando comparados com os 23% que deram a mesma resposta na pesquisa de 2019. Já 64% consideram a blockchain como uma alternativa mais segura que as soluções convencionais da TI, e 31% consideram ter o mesmo nível de segurança que as soluções existentes.

A tecnologia blockchain pode ser aplicada em diferentes casos de uso, em especial naqueles em que exista uma necessidade explícita de se garantir a confiança na realização de transações entre duas ou mais partes, sem a necessidade de etapas intermediárias de verificação — o que economiza recursos —, e em que seja possível identificar de forma inequívoca a origem dos dados transacionados, garantindo a privacidade dos agentes, ao passo que permite seu monitoramento em tempo real.

É o caso, por exemplo, do monitoramento da cadeia de custódia (*supply chain*), indústria que dá grande destaque às aplicações de blockchain em atividades de controle e certificação de origem de produtos e no acompanhamento do transporte de mercadorias. Uma vez que os dados em uma cadeia blockchain são imutáveis, esta se torna uma tecnologia extremamente sensível na prevenção a fraudes e outros desvios.

Outro exemplo de caso de uso que utiliza a blockchain como tecnologia é a indústria de meio de pagamentos e de transferência de dinheiro. Com seu sistema de transações seguras, a blockchain permite a transferência monetária de forma quase que instantânea de uma ponta à outra, sem passar por agentes intermediários como agentes financeiros e governos, não estando sujeita, por exemplo, ao pagamento de taxas e impostos sobre operações nem dependente de tempos e prazos de compensação, funcionando 24 horas por dia, todos os dias.

A manutenção de documentos digitais é outro exemplo bastante clássico do uso da blockchain, e aqui falamos não apenas de documentos de identificação (cuja veracidade é assegurada pela garantia de origem e imutabilidade) como, ainda, na autenticação e validações tais como as realizadas por cartórios no Brasil. Sistemas de votação também entram nesta categoria, considerando as necessidades de apuração e contagem em tempo real, da imutabilidade, e da segurança das transações, com garantia da origem e privacidade do voto depositado.

A indústria da saúde também vem se beneficiando da blockchain, por exemplo, na manutenção de prontuários médicos seguros, em que diversos documentos sobre determinada pessoa são armazenados, podendo ser acessados de qualquer ponto da rede com garantia de privacidade. Uma aplicação curiosa, que une a saúde com a manutenção da cadeia de custódia, é a da dispensação de remédios controlados que, quando gerenciados por meio de uma blockchain, só podem ser vendidos e entregues ao titular da transação e a ninguém mais. Essa garantia de identificação é fundamental para aumentar ainda mais a segurança do processo.

Exemplos de uso de blockchain incluem ainda casos na indústria do entretenimento, em segurança alimentar, indústria automobilística, no ramo imobiliário, no meio educacional (para validação de diplomas, por exemplo), em serviços governamentais, entre muitos outros.

Além de serviços específicos para indústrias, hoje já encontramos uma grande gama serviços disponíveis para serem utilizados por usuários finais da Internet, com aplicações que vão desde o armazenamento de arquivos em blockchain, redes sociais, comunicadores instantâneos, plataformas para a realização de videochamadas e, até mesmo, sistemas operacionais inteiros!

1.4 Blockchain-as-a-service (BaaS)

O conceito de *Blockchain-as-a-Service* ("Blockchain como Serviço") vem ao encontro de definições mais modernas e populares, como o IaaS, o PaaS e o SaaS — significando, respectivamente, a oferta de infraestrutura, plataforma e software como serviços — que ganharam notoriedade especialmente com o advento da computação em nuvem. Nesses modelos, uma empresa ou organização possuidora de uma infraestrutura, plataforma ou software disponibiliza sua capacidade para ser utilizada por usuários finais em troca de uma quantia de dinheiro cobrada, normalmente, por tempo de uso.

A grande vantagem do uso desse modelo de negócio para os usuários-consumidores é a de diminuir os custos iniciais necessários para o projeto, evitando, por exemplo, a compra de equipamento e preparação da infraestrutura, além de ajudar a controlar os custos com base na capacidade requerida. Isso previne, ainda, o fenômeno da subutilização de recursos e permite otimizar o tempo de início da operação, evitando grandes configurações e preparo de ambiente, entre outros.

Seguindo a mesma linha de raciocínio, o BaaS já vem sendo ofertado por importantes atores do mercado de tecnologia. Por definição, o BaaS realiza a oferta de uma plataforma de blockchain gerenciável que permita aos usuários construírem aplicações e serviços digitais em uma rede distribuída, ao passo que a

infraestrutura e o ferramental de construção da cadeia são oferecidos pelo proprietário.

A *Amazon Web Services* (AWS), líder do segmento de Computação em Nuvem segundo o Quadrante Mágico da Gartner publicado em agosto de 2020, por exemplo, anuncia em sua documentação oficial possuir mais de 70 soluções dedicadas à blockchain que envolvem bancos de dados especializados em registros de transações e serviços gerenciados — como o *Amazon Managed Blockchain* — que permitem a criação de redes de blockchain privadas sobre estruturas de redes blockchain públicas.

Da mesma forma, a Azure, da *Microsoft* ofertou o Azure BaaS até 2021 e, logo depois, seus serviços foram migrados para o *Quorum Blockchain Service*, (QBM) da ConsenSys, hospedado totalmente na infraestrutura da *Microsoft*. Sua solução é produtizada como uma suíte de produtos que envolvem ambientes de desenvolvimento e teste, módulos de segurança da rede, um conjunto de APIs de acesso à rede blockchain, entre outros, de forma tornar mais eficiente o desenvolvimento de soluções em redes distribuídas.

Um ator extremamente importante neste meio, o *Hyperledger*, também deve ser citado. Trata-se de um projeto *open-source* sob tutela da *Linux Foundation* lançado em 2015 com a missão de criar um ecossistema de desenvolvimento blockchain a ser utilizado como incubador para as empresas que desejam apostar nesta tecnologia no âmbito privado (normalmente utilizado por empresas e organizações).

Através de uma filosofia de desenvolvimento modular, altamente segura, interoperável, agnóstica para o uso de criptomoedas e trabalhada por meio de APIs especializadas, o *Hyperledger* fornece uma série de frameworks e ferramentas de trabalho que atendem às necessidades de desenvolvedores, respeitando a individualidade de cada tipo de indústria. Entre eles, destacam-se:

1. O *Hyperledger Fabric* — base de desenvolvimento do projeto, ele modulariza os principais componentes necessários para um desenvolvimento blockchain tais como a criação de redes, a execução de transações ponto a ponto, definição dos protocolos de consenso e serviços de privacidade, todos usando uma filosofia *plug-and-play*.

2. O *Hyperledger Sawtooth* — com ênfase no desenho de aplicações distribuídas, ele trabalha com o desenvolvimento, configuração, parametrização e com o *deploy* dos contratos inteligentes — que como veremos nos próximos capítulos são, efetivamente, os programas desenvolvidos e executados na rede blockchain.

3. O *Hyperledger Iroha* — framework com foco no desenvolvimento de soluções para IoT e soluções mobile que utilizem uma rede blockchain como base. Ele gerencia os participantes da rede e permite a interação entre os pontos, controlando, por exemplo, as permissões de acesso e o acesso a dados, de forma muito rápida.

4. O *Hyperledger Caliper* — utilizado como uma ferramenta de *benchmark* e de coleta de informações sobre as aplicações blockchain. Ele executa a funcionalidade de monitoramento de performance e de geração de relatórios sobre as aplicações de interesse.

5. O *Hyperledger Explorer* — ferramenta utilizada para a visualização e busca de dados em blocos, transações e metadados de rede.

O projeto *Hyperledger* é usado como base para a oferta de soluções de empresas importantes como a *Oracle*, através do *Oracle Platform* e a *IBM* com a *IBM Blockchain Platform*.

Um segundo projeto extremamente importante para a comunidade, o *Corda* da R3, é uma plataforma apresentada como uma solução *open-source* para a criação de aplicações em blockchain, também

com ênfase no desenvolvimento para indústrias e organizações. Ele permite a criação de contratos inteligentes, hospedados na rede para a criação de aplicações distribuídas (chamadas de *CordApps*).

No ecossistema Corda, atualizações feitas ao estado de suas aplicações são tratadas como transações e, como tais, incluídas na blockchain através de seus mecanismos de funcionamento.

Assim como na Computação em Nuvem trabalhamos com nuvens públicas, privadas e híbridas, no universo Blockchain também encontramos diferentes classificações. São elas:

1. Redes Públicas — possuem participação aberta para qualquer tipo de ator; suas aplicações têm um uso geral e público, por exemplo, no trabalho com criptomoedas.
2. Redes Privadas — são redes administradas por uma organização, que controla quem possui permissão para participar e executar os protocolos de consenso.
3. Redes Autorizadas (permissionadas) — podem ser Públicas ou Privadas, mas são mais restritas do que as suas congêneres. Aqui, podem ser impostas políticas de restrição à participação na rede e também de permissões para a execução de transações.
4. Redes de Consórcio — são redes compartilhadas por mais de uma organização, entre as quais existe divisão de responsabilidades e permissões.

A importância do BaaS pode ser medida como resultado do porte das organizações envolvidas em sua oferta e como membros apoiadores de projetos de desenvolvimento sendo, sem dúvidas, um serviço extremamente importante, seguro e confiável.

Existem críticas por parte de comunidade de que a utilização do BaaS violaria o princípio do blockchain de não possuir pontos centrais de controle — e que neste caso seriam desempenhados pelas empresas fornecedoras do serviço. Ainda que a crítica faça sentido, há que se pesar aqui os ganhos de performance e eficiência obtidos ao optar pelo seu uso, que são inegáveis.

1.5 Afinal, o que é a Rede Ethereum?

A Rede Ethereum é uma blockchain pública não permissionada, de uso geral, idealizada por Vitalik Buterin, existente desde 2015 e que possui a capacidade de armazenar e executar aplicações desenvolvidas nativamente para a tecnologia da cadeia de blocos de forma descentralizada, através dos chamados “contratos inteligentes” (*smart contracts*). Ela possui sua criptomoeda nativa, o Ether, que é um dos maiores ativos reconhecidos em mercado e valor.

A grande jogada de Vitalik Buterin ao sugerir o Ethereum foi a de visar criar um grande “computador mundial”, com uma infraestrutura e capacidades inalcançáveis por equipamentos comuns, resultado do compartilhamento de recursos computacionais de todos os participantes da rede, inexistindo a possibilidade de quedas de sistema e ações de censura, fraude ou interferências externas — isto, aproveitando-se dos benefícios da própria estrutura da tecnologia blockchain.

O Ethereum não apenas é uma rede, mas também pode ser visto como um grande banco de dados transacional e também como um único grande computador, residindo sob a *Ethereum Virtual Machine* (EVM).

As aplicações existem no Ethereum na forma de *smart contracts* e são replicadas ao longo da rede, sendo absolutamente imutáveis:

uma vez que o contrato é enviado para a rede, ele não pode ser alterado. Os dados de transações são armazenados em cada um dos blocos pertencentes à cadeia e, por estarem distribuídos ao longo de todos os participantes da rede, também são totalmente seguros.

Maiores detalhes sobre o funcionamento da Rede Ethereum serão dados nos capítulos posteriores, e incluirão detalhes, por exemplo, sobre mineração, algoritmos de consenso, uso de *tokens*, entre outros.

A Rede Ethereum é hoje o principal suporte para as aplicações blockchain existentes, com crescimento da ordem de 65% no primeiro semestre de 2021. Neste mesmo ano, o Ethereum ultrapassou o Bitcoin em número de endereços ativos, muito disto devido à sua predominância sobre o mercado de *tokens* não fungíveis (NFTs) e, mais recentemente, ao mercado de finanças descentralizadas (DeFi) — protocolos interoperáveis através do qual serviços financeiros são disponibilizados para o público sem que haja figuras intermediárias, como bancos e organizações.

Conclusão

A blockchain, como vimos, é uma tecnologia que desperta muita expectativa, e sobre a qual devemos esperar grandes evoluções em um futuro próximo. Prova disso são as magnitudes de atores do mercado da tecnologia que, de alguma maneira, estão envolvidos com ela — seja utilizando em projetos próprios, seja ofertando serviços dedicados — o que demonstra seu potencial.

Ainda que seu nome possa indicar ser uma tecnologia coesa e uniforme, esta é, na realidade, uma impressão bastante equivocada. A própria bibliografia especializada é extremamente diversificada, com publicações que lidam desde temas gerais como arquiteturas de aplicações blockchain, que dão ênfase a decisões sobre estrutura e seleção de protocolo em fases iniciais de projeto, até o uso de blockchains específicas para a solução de cenários bem

contextualizados, e parte da magia de se trabalhar com essa tecnologia está aqui: temos um universo de possibilidades a serem exploradas!

Ao longo do livro daremos ênfase à utilização da blockchain Ethereum, que é um dos maiores expoentes desta área — usada como base para o desenvolvimento de aplicações públicas descentralizadas e distribuídas em contextos gerais. Seu desenvolvimento foi feito com a ideia de aproveitar a "estrutura blockchain", porém, trazendo para ela capacidades computacionais.

Se nos lembrarmos agora do conceito da *Turing Completeness* ("Turing completude") — e sobre o qual falaremos mais a frente —, dizemos que o Ethereum é "*quase*-completa", sendo capaz de executar um rol *quase* ilimitado de instruções lógicas; daí seu grande poder.

Ao longo dos próximos capítulos veremos detalhes sobre o funcionamento desta tecnologia tão fascinante, que certamente trará vantagens competitivas importantes para quem a explorar.

Capítulo 2
Blockchain para principiantes

2.1 Introdução

No capítulo anterior, vimos que a tecnologia blockchain, tal como conhecemos hoje, surgiu em 2008, após uma publicação atribuída a Satoshi Nakamoto, que apresentava a ideia do Bitcoin como um *sistema de transações digitais* baseado em uma rede P2P. Abordamos também o fato de sua ideia central ser a de existir como um grande "livro de transações" (às vezes chamado Livro Razão, por alusão a uma importante ferramenta da Contabilidade), conhecido tecnicamente pela expressão em inglês *ledger*, distribuído na rede.

Em uma rede deste tipo as transações (isto é, as operações realizadas nela) são armazenadas em blocos de informação, ligados uns aos outros através de mecanismos seguros, que garantem a inviolabilidade e a imutabilidade dos dados ali contidos. Desta união dos blocos de transação advém a ideia da "cadeia de blocos", que dá o nome à tecnologia — e aí a expressão "blockchain".

De forma geral uma rede P2P funciona quando dois ou mais computadores — chamados de nós — se conectam uns aos outros com o fim de compartilharem recursos, sem que exista a figura central do administrador de rede, com papel de gerenciador deste ambiente. Por essa característica, diz-se que ela é descentralizada.

Na rede P2P cada um dos nós possui capacidades e responsabilidades compartilhadas, inclusive a de coexistirem como clientes e servidores ao mesmo tempo. Sua capacidade de processamento e sua velocidade são proporcionais ao seu tamanho (ao número de participantes). Este é o princípio de funcionamento, por exemplo, do protocolo BitTorrent, bastante conhecido e utilizado para o compartilhamento de arquivos através da Internet.

Na blockchain, cada nó presente na rede possui uma cópia exata de todos os blocos de transações existentes para aquela cadeia, inexistindo uma fonte única de dados. Daqui advém sua segunda característica de grande importância: é uma rede distribuída, em que cada nó é igual aos demais.

Unidas, a descentralização e a distribuição provêm à blockchain o reconhecimento como uma tecnologia segura, já que, inexistindo uma fonte única de verdade, tal como um repositório, caso algum nó falhe ou sofra alguma alteração de forma proposital, as outras cópias presentes na rede são capazes de identificar e rejeitar o comportamento desviante, restaurando o nó alterado à sua versão original. A isso, soma-se o fato de que cada bloco de transações possuir identificadores criptográficos dele e do bloco anterior a ele. Caso qualquer dado seja alterado em um ponto da cadeia, o valor criptografado é alterado, tornando o restante da rede inválido, o que é capaz de disparar ações de alerta e reparo.

Essas características garantem à blockchain vantagens como a impossibilidade de censura e/ou o controle por organizações, já que qualquer alteração (ainda que fosse o desligamento do sistema) necessitaria ser realizada na grande maioria dos nós da rede ao mesmo tempo, o que é uma tarefa que beira o impossível, e dá a ela o poder de garantir que seus dados sejam imutáveis.

Se a alteração dos dados é uma ação impossível na rede em condições normais, a inclusão de novos blocos de transação, por outro lado, deve ser uma tarefa corriqueira dentro dela — e isso é uma condição para seu funcionamento. Visto ser impossível fazer a adição de um bloco em todos os nós de forma simultânea e considerando que, por possuírem os nós os mesmos papéis e responsabilidades, qualquer um deles deve ser capaz de acrescentar um novo bloco a qualquer momento, deve haver um mecanismo que possibilite que eles entrem em um acordo sobre como aceitar (ou não) esta mudança na cadeia.

Esse é o papel dos *algoritmos de consenso* e, a solução deles é realizada por meio dos *mineradores* ou *validadores* — participantes da rede que se propõem a resolver problemas complexos, recebendo como recompensa um valor determinado de criptomoedas.

A seguir veremos como cada um destes componentes funciona, o que permite o funcionamento da blockchain como um ecossistema orgânico, em constante crescimento.

2.2 Bloco de transação e a Árvore Merkle

A esta altura, falar que o componente principal de uma blockchain é o “bloco” deveria ser redundante. A formatação dos blocos que compõem a blockchain é variável de acordo com a rede em uso — e as especificidades da Rede Ethereum serão dadas no próximo capítulo — porém, alguns elementos são básicos e comuns a todas as cadeias, e envolvem a existência do *bloco* como uma *estrutura*.

Um bloco, isoladamente, é constituído por um cabeçalho, que contém sua identificação e configurações, e pela área destinada a guardar a lista de transações contidas.

No cabeçalho são inseridas suas informações de identificação, tais como numerações sequenciais para informar sua a posição, informações sobre versão e sua data e hora de criação. Além disso, ele possui duas informações de segurança importantíssimas: o *hash* criptográfico do bloco imediatamente anterior, e o seu próprio *hash*.

Maiores informações sobre criptografia serão dadas no próximo tópico, porém, para o momento basta saber que um *hash* age como um identificador inequívoco de determinado conteúdo — no caso, pode-se entendê-lo como a “impressão digital” do bloco como um todo.

Em uma blockchain típica, o cabeçalho do bloco é composto por uma String de 80 bytes, sendo 4 bytes usados para a sua identificação, 32 bytes usados para armazenar o *hash* do bloco anterior, 32 bytes guardando o *hash* do bloco atual, 4 bytes dedicados a guardar a data e hora de sua criação, e mais 8 bytes usados durante o processo de mineração — sendo 4 bytes representantes da dificuldade da mineração e os outros 4 bytes contendo um valor denominado *"Nonce"*, que representa o resultado do trabalho do minerador.

O primeiro bloco de uma blockchain é chamado de **Bloco Gênese** (*Genesis Block*), ou "Bloco 0", e é diferenciado dos demais por não possuir um valor de *hash* indicando o bloco anterior.

O verdadeiro significado do armazenamento do *hash* do bloco anterior é, na realidade, uma alusão ao funcionamento de uma estrutura de dados chamada de **Árvore Merkle**, que permite a checagem de grandes quantidades de dados organizados, ao relacionar seus identificadores em uma única estrutura, em forma de árvore.

Proposta em 1979 por Ralph Merkle, essa estrutura agrupa nós de forma hierárquica, nos quais os identificadores exclusivos de cada nó-pai são gerados a partir dos identificadores exclusivos de seus nós-filhos correspondentes, de forma repetida, até que se chegue à raiz — chamada *Raiz Merkle*. Esta raiz torna-se identificada por um identificador exclusivo gerado a partir dos identificadores exclusivos de cada um dos nós da árvore. Em outras palavras: o *hash* atual é calculado, diretamente, com base em todos os *hashes* anteriores existentes na rede, indiretamente.

A grande vantagem da utilização de uma Árvore Merkle em um cenário em que exista uma grande quantidade de identificadores, tal como em uma blockchain, é a de que não existe a necessidade se de recalcular cada um dos identificadores exclusivos de cada um dos nós sempre que um dado deva ser verificado (seja na adição ou na edição — inexistente, no caso da blockchain).

Uma vez que qualquer alteração em um bloco altera seu *hash* — isto é, seu identificador exclusivo —, o seu relacionamento com os outros nós da árvore é invalidado imediatamente, indicando aí uma alteração que não deveria ter ocorrido e, portanto, tornando inválida a transação.

Graças ao uso da Árvore Merkle a blockchain é uma tecnologia que provê a imutabilidade de seus dados já que, para que um bloco existente fosse adulterado, seria necessário adulterar os demais blocos subsequentes também, garantindo que o novo *hash* fosse propagado ao longo dos nós até a raiz; um processo, no mínimo, trabalhoso.

Junta-se a isso o fato de que, na blockchain, existe um fator de dificuldade incluído no cálculo do identificador exclusivo, o que torna a tarefa de alterar um grande número de blocos simultaneamente praticamente impossível. Além disto, ainda existe o fato de ela ser uma rede distribuída, e que a mesma ação maliciosa deveria ser realizada em todos os nós da rede ao mesmo tempo.

2.3 Criptografia e validação

Até agora os termos *“hash”* e “identificador exclusivo” foram utilizados livremente, como se fossem intercambiáveis. Na verdade, assim o são, realmente.

Um *hash* pode ser definido tecnicamente como o resultado da aplicação de uma função criptográfica usada para mapear dados, de tamanho variável, com uma saída de tamanho fixo.

Anteriormente, quando mencionamos um espaço de 32 bytes reservado para cada um dos *hashes* do bloco, por exemplo, nos referíamos à aplicação de uma função criptográfica geradora de *hash* chamada SHA-256, que tem como característica gerar como saída um *hash* com tamanho fixo de 32 bytes (ou 64 caracteres,

considerando a representação notação hexadecimal), independentemente do tamanho do dado sobre o qual é aplicada a criptografia.

Este tipo de função criptográfica é considerado sendo “de via única”, isto é: ela gera um *hash* conhecido, porém, a partir dele é impossível identificar o dado original.

Funções criptográficas de geração de *hash*, no geral, estão sujeitas às seguintes condições de validade para serem aceitas em uso corrente:

1. Determinismo — uma mesma entrada sobre a qual é gerado um *hash* criptográfico deve sempre produzir exatamente o mesmo resultado;
2. Não correlacionabilidade — uma pequena alteração no dado de origem deve ser capaz de promover uma mudança tão grande no *hash* que seja impossível correlacionar as mudanças nos dados com os efeitos causados durante a operação de *hashing*;
3. Irreversibilidade — deve ser impossível identificar o dado de entrada a partir do *hash* conhecido;
4. Verificabilidade — o cálculo do *hash* deve ser eficiente e computacionalmente possível;
5. Proteção a colisões — não deve ser factível que duas entradas diferentes produzam o mesmo *hash*.

O SHA-256, mencionado anteriormente, é um dos métodos de criptografia mais utilizados. Seu resultado é um *hash* de 64 caracteres em notação hexadecimal capaz, portanto, de produzir 256 caracteres diferentes em cada um de seus bits — um total de combinações de 2^{256} (ou, aproximadamente, 1077 *hashes* diferentes).

Ainda que, em teoria, seja possível haver colisões, esta é uma possibilidade com probabilidade quase nula. Ataques do tipo de

força bruta, em que computadores são programados para tentar todas as combinações possíveis, também são improváveis de funcionarem: estima-se que um computador normal levaria mais anos tentando encontrar uma única colisão de um único *hash* do que o número de anos que nosso universo possui!

Como exemplo, considere o *hash* para a mensagem de entrada “Olá, mundo”, utilizando o SHA-256:

```
b8a0136dcb2459c3070d390239f18258248d9f9d866f31cc822d308620b90882
```

O mesmo *hash* para a expressão “Olá, mundo!” (agora com um "!" ao final) é:

```
9583b013bà520d3a893c4270d0c67732d7ef1768eb0a13533b4e7b134d4b131
```

Observa-se que uma pequena alteração, que foi a inclusão de um ponto de exclamação ao final, produziu um resultado totalmente diferente.

Como veremos mais à frente, a Rede Ethereum utiliza uma função criptográfica para geração de *hash* baseada no padrão Keccak-256, que possui suas peculiaridades e é ainda mais estrita que o SHA-256. Para exemplo de comparação, o *hash* para a mesma expressão “Olá, mundo” por essa outra função criptográfica é dada a seguir e, como é fácil notar, apresenta um resultado totalmente diferente daquele conseguido pelo SHA-256:

```
31494269f7e2476375624f6a346ab854045fed31f07341d6ce8a0c6db1ba98d0
```

2.4 Protocolos de consenso

Os protocolos de consenso surgem como uma ferramenta de controle da blockchain. Quando um novo bloco é adicionado à cadeia, os protocolos de consenso agem para definir as regras de validade de um bloco e orquestrar o crescimento da cadeia. Afinal,

quando um bloco é adicionado, esta informação precisa ser repassada para todos os nós da rede e, por mais rápido que isso ocorra, é possível que mais algum nó consiga incluir um novo bloco à cadeia exatamente neste intervalo — o que causaria um conflito entre nós existentes.

Um segundo motivo para este controle diz respeito à proteção contra ataques maliciosos que sejam capazes de inserir um bloco ao final da cadeia em algum dos nós; nesta situação, a rede deve ser capaz de se proteger, identificando-o como uma ameaça e o rejeitando.

Esse último caso é mais simples de ser solucionado: através de uma série de regras definidas pelo protocolo de consenso determina-se se o bloco inserido atende aos requisitos preestabelecidos em uma espécie de “contrato”. Essa mesma lista de requisitos é verificada em cada um dos nós, sendo repetida, portanto, várias vezes.

O caso do conflito, no entanto, em que dois nós incluam dois blocos válidos simultaneamente ou em períodos de tempo muito próximos, em que ambos atendam a todos os requisitos de segurança, é mais complicado e precisa ser resolvido por meio da aplicação de diferentes técnicas.

São dois os principais protocolos de consenso normalmente utilizados: o *Proof-of-Work* (PoW) — prova de trabalho — e o *Proof-of-Stake* (PoS) — prova de participação. Estes dois tipos de algoritmo definem métodos, através dos quais um bloco é considerado válido, estando apto para ser adicionado à cadeia. Eles também garantem as condições para sua inclusão de forma organizada, mantendo a integridade da blockchain.

Apesar de serem os dois principais protocolos de consenso, não são os únicos. Um exemplo é o *Proof-of-Authority* (PoA), utilizado em mecanismos de escalabilidade da Rede Ethereum, que também será mencionado mais à frente, neste livro.

Tanto no PoW como no PoS existe a possibilidade de que dois blocos válidos diferentes sejam adicionados ao mesmo tempo a diferentes nós, ainda que ambas as técnicas sejam estritas e trabalhem, justamente, para evitar este tipo de situação. Nesse caso, em algum momento e, por definição da blockchain em questão, há que se optar por uma das cópias da cadeia que contenham algum dos blocos válidos conflitantes. Uma opção comum é a de esperar que mais um bloco seja minerado; a cadeia que receber o próximo bloco em primeiro lugar torna-se a dominante, por ser maior (entendida também como mais "estável"), vencendo a disputa. O bloco válido rejeitado subsiste no ecossistema, porém, fora da cadeia principal.

Considerando que as informações principais de um bloco, além da sua identificação e da lista de transações, são os *hashes* — do bloco imediatamente anterior e do bloco em questão — a adição de um novo bloco à cadeia, em tese, deveria ser uma tarefa simples, uma vez que o cálculo do *hash* é algo extremamente rápido de ser feito via programação.

Acontece que, em uma situação em que em determinada rede existe um número muito grande de nós participantes e, considerando que todos sejam capazes de incluir blocos a qualquer momento — como é o caso de uma blockchain —, esta situação seria extremamente difícil de ser controlada.

Considerando ainda que o bloco precisaria ser adicionado, replicado para todos os nós participantes em um período de tempo extremamente curto, que a ordenação dos blocos deve ser lógica e que uma transação deve ser levada para a cadeia uma única vez, isso se torna um trabalho hercúleo.

Uma solução fácil para isso seria a criação de um agente orquestrador, que definiria como as transações seriam divididas e saberia quais nós teriam quais pedaços de informação, para conseguir fazer todos trabalharem em sinergia. Essa é uma solução

muito comumente utilizada em sistemas distribuídos e que comprovadamente funciona.

O grande problema aqui é que a blockchain é, por definição, um sistema descentralizado e, portanto, a inclusão de um agente centralizador destes feriria um de seus principais pilares. Não obstante, em algum momento essa implementação necessitaria de intervenção humana para, por exemplo, controlar parâmetros ou atualizar a implementação para se adequar à carga de trabalho. Levando em consideração que em uma blockchain as instruções são imutáveis, isso também não funcionaria aqui.

Para contornar esses problemas é que existem os algoritmos de consenso. São instruções de execução e avaliação de conhecimento público, executadas de forma idêntica por todos os nós, e que são capazes de tomar estas decisões, como veremos a seguir.

Proof-of-Work — Mineração

A “mineração” em blockchain surge como uma alternativa para resolver o problema do conflito de blocos, incluindo um grau de dificuldade para que o bloco seja incluído e desencorajando seu uso malicioso.

Nela, os nós participantes (isto é, participantes ativos da rede — os mineradores) competem entre si para solucionar um desafio lógico. O primeiro a conseguir tal façanha é o minerador que define o próximo bloco a ser inserido na cadeia e, como recompensa, recebe uma quantia em criptomoedas de volta que inclui, ainda, uma contribuição dada por cada uma das transações inseridas no bloco.

Caso o bloco seja incluído, porém seja rejeitado pela rede em seguida (como no caso de uma inclusão fraudulenta), além de ele ser invalidado, a quantia em dinheiro deixa de ser paga para o nó que solucionou aquele desafio, o que traz perdas financeiras para o minerador, visto o processo ser custoso.

No Bitcoin, que serve de base para muitos projetos de blockchain, o objetivo dos mineradores é o de identificar o valor ótimo do *Nonce* — expressão gerada a partir de *number* (número) e *once* (uma vez) —, um valor numérico de 4 bytes que, quando incluído nas informações do bloco, gera um *hash* criptográfico que atende às condições especificadas — o chamado *target* (alvo).

Sendo o único valor variável dentro do bloco, ao ser alterado, o valor do *Nonce* provoca uma atualização no *hash* do bloco que se deseja incluir de forma automática. Na Rede Ethereum o termo *Nonce* aparece em mais de uma oportunidade, com finalidades um pouco diversas, como será visto posteriormente.

Aqui acabamos de definir o chamado "quebra-cabeças lógico": um desafio computacional que deve ser resolvido pelos mineradores de forma a validar ou não a inclusão do bloco na rede.

Considere o *hash* obtido previamente para a frase “Olá, mundo”; seu resultado é um número expresso na notação hexadecimal que varia entre o valor em que todos seus algarismos são iguais a 0, até o valor em que todos seus algarismos são iguais a F. Um desafio lógico comum da blockchain é o de se definir um número-alvo máximo a ser atingido e estabelecer a condição de que o *Nonce* calculado seja menor que ele.

Exemplificamos: em um caso hipotético de uma blockchain que trabalha com um tipo de *hash* muito específico que sempre resulta em um número entre 0 e 100 e determinamos que o target da cadeia é o número 25. Os mineradores teriam como missão calcular um valor de *Nonce* que, ao ser combinado com as informações do bloco a ser adicionado e tratado por uma função de *hash* criptográfico, resultaria em um valor menor do que 25. O minerador que encontrasse a resposta 24 ou a resposta 5, por exemplo, venceria a competição — apenas o primeiro a apresentar a resposta vence a competição —, enquanto os mineradores que encontrassem valores iguais a 70 ou 30 não teriam suas respostas aceitas.

Considerando que um *hash* SHA-256, por exemplo, tem 64 caracteres, isso é um pouco mais complicado — e ainda vai ficar pior!

Uma forma comum de se expressar o *target* é o de se definir a quantidade de zeros à esquerda do número — por exemplo, o valor 003 tem dois zeros à esquerda, enquanto o valor 000003 tem cinco.

Considerando nosso exemplo do *hash* que gera valores entre 0 e 100, um target de dois zeros aceitaria apenas valores entre 000 e 009, enquanto um *target* de um zero aceitaria valores entre 000 e 099 — o que aumenta substancialmente as probabilidades de acerto. Transporte isso para nosso *hash* hexadecimal de 64 caracteres: cada zero à esquerda reduz em 16 vezes o intervalo de valores que podem ser aceitos.

Nosso *hash* de 64 caracteres pode gerar até 1664 *hashes*, aproximadamente 1077 valores diferentes. Uma regra da minha blockchain que determine um *target* de 10 zeros à esquerda, por exemplo, reduz o número de *hashes* possíveis para apenas $1664^{?1°} = 1654$ ou, aproximadamente, 1065 *hashes* possíveis. Parece um número grande, porém, analisando as probabilidades, isso significa que, ao escolher um número aleatório, temos uma probabilidade de acerto de 1065/1077 — apenas 0,0000000001%!

Esse parâmetro de dificuldade é controlado dinamicamente, sem intervenção humana, através de definições feitas no início da blockchain. No caso do Bitcoin, por exemplo, ele é ajustado a cada 2016 blocos inseridos (aproximadamente duas semanas), de forma a manter o tempo médio de mineração de cada bloco próximo a 10 minutos — por isso, diz-se que o tempo de transação do Bitcoin é de 10 minutos.

Na Rede Ethereum, atinge-se o valor de 1 bloco adicionado a cada 15 segundos, em média; isso por definições do próprio projeto. Esses valores não têm uma implicação imediata ou outros motivos, além do controle do tempo de crescimento da cadeia.

O *Nonce* é um valor de 4 bytes do tipo *unsigned* (não aceitando, portanto, valores negativos), o que permite um intervalo de valores inteiros entre 0 e 4.294.967.295. Este valor não é absurdo de ser atingido por computadores muito potentes, muito menos em um cenário de processamento distribuído em que diferentes nós podem testar diferentes intervalos aí dentro; assim, em tese todos os valores possíveis poderiam ser testados até que um deles fosse reconhecido como um *Nonce* válido.

Diminuindo o intervalo de possibilidades, nossa probabilidade de acerto de 0,0000000001% aumenta um pouco, porém ainda representa um número muito pequeno — e aqui consideramos apenas 10 zeros à esquerda para fazer estes cálculos — existindo, portanto, cenários em que mesmo que todos os valores de *Nonce* possíveis sejam testados, nenhum deles atinge o *target* desejado. Nesse caso, a opção é alterar o bloco de transações sendo inseridas ao bloco, que além do *Nonce* é a única outra sessão do bloco que pode variar, iniciando uma nova rodada de tentativas e erros.

O processo de mineração é ainda mais complexo que isso. Não podemos esquecer que dentro do bloco existe também um campo reservado para armazenar a data e a hora de criação — que já havia sido previsto na primeira publicação de Stuart Haber e W. Scott Stornetta. Esse campo considera o momento em que o bloco é minerado, e, por conter menção à hora de criação, é também atualizado constantemente, a cada segundo.

Assim, os cálculos dos *hashes* possíveis para aquele bloco são válidos dentro do período de um único segundo; após isso precisam ser recalculados novamente, reiniciando todo o processo. Em outras palavras: todos os cálculos de *Nonce* devem ser feitos em 1 segundo, e por isto é tão difícil.

O processo de mineração é, como visto, extremamente desafiador, e exige grande eficiência computacional. Não é difícil encontrar na Internet imagens sobre verdadeiras indústrias de mineração de

criptomoedas! A recompensa vale o esforço: o Bitcoin, por exemplo, premia o primeiro minerador a encontrar o valor do *Nonce* com 12,5 bitcoins, atualmente, mais as taxas de cada uma das transações — isto, após o bloco ter sido aceito pela cadeia como um bloco válido. Este valor, dependendo da cotação da moeda virtual, facilmente ultrapassa 1 milhão de reais por um único bloco minerado. Ao longo do tempo, de acordo com a política monetária do Bitcoin, a cada 210.000 blocos minerados essa quantidade de recompensa é reduzida pela metade.

Proof-of-Stake — Validação

Ainda que o termo “mineração”, quando aplicado a criptomedas, seja mais comumente utilizado, o protocolo de consenso por participação — o PoS — é, surpreendentemente, mais antigo que o PoW e tem recebido bastante atenção nos últimos anos devido às inovações tecnológicas aplicadas a ele.

Em uma blockchain que utiliza o PoS, existe o papel de um usuário (ou nó) validador: são validadores aqueles que possuem um determinado valor poupado da criptomoeda correspondente à cadeia sendo utilizada (Ether, no caso da Rede Ethereum) e que geram um tipo específico de transação, indicando que desejam validar o bloco em questão. Para isso, depositam uma parte de seus fundos.

Quando um quórum de validadores é atingido, o chamado “comitê”, ele recebe o direito de opinar sobre qual o próximo bloco a ser incluído na cadeia e, posteriormente, faz sua avaliação, através de protocolos de domínio público. Todo esse processo deve ocorrer dentro de um intervalo de tempo determinado, chamado de *slot* (ou “fenda”, em português). Durante o *slot*, havendo um consenso entre os validadores, o bloco em questão é aprovado e incluído à cadeia.

Um único bloco pode ser incluído durante um slot. Um conjunto de *slots* é chamado de “época” (do inglês, *epoch*). Um comitê é válido

durante uma época, e logo depois é desfeito — dando chances para novos participantes entrarem como validadores.

Existem critérios bem definidos de precedência de validação dentro do PoS. Por exemplo, os nós que possuam maior quantidade de criptomoedas depositadas votam com um peso maior.

Caso o bloco em avaliação seja aprovado para ser incluído à blockchain, os nós validadores que votaram positivamente para ele recebem seu depósito de volta e mais uma quantidade monetária proporcional ao valor depositado originalmente. Caso contrário, isto é, caso o bloco seja rejeitado, os validadores que tenham votado a favor dele perdem o valor depositado, e não recebem nada em volta. Essa é uma forma interessante de garantir que todos os participantes ajam de forma honesta, já que são penalizados financeiramente no caso de não fazerem uma boa avaliação.

O PoS é considerado por muitos autores como um método menos maduro que o PoW. Apesar disso, ele é utilizado por blockchains importantes, com bastante êxito, tanto por sua estrutura de trabalho bem definida, como por trazer vantagens. Entre elas:

1. Menores custos associados a equipamentos e estrutura física;
2. Maior velocidade, permitindo menores tempos de transação;
3. Menor gasto energético — sendo um método mais correto, ecologicamente;
4. É um método mais descentralizado que o PoW, já que vários nós participam simultaneamente na mesma decisão.

Grandes projetos iniciaram suas cadeias utilizando o PoW como consenso e posteriormente migraram, ou planejam migrar, para o PoS. É o caso da própria Rede Ethereum, cuja implementação de PoS, chamada Casper, que será apresentada em momento adequado, é muito esperada pela comunidade.

Proof-of-Authority — Validação

O protocolo de consenso de *Proof-of-Authority* (PoA) é similar ao PoS, sendo considerado, por muitos, como uma evolução deste.

Em um ambiente com uso de PoA existe um número seleto de validadores (entre 20 e 30 participantes, apenas), e seu processo de validação segue a mesma ideia: validadores apresentam sua decisão sobre a validade ou não das transações e do bloco como um todo e entram em um consenso sobre isto.

Diferentemente do PoW e do PoS, em que as atividades estão abertas para qualquer indivíduo ou organização que queira atuar como um nó, no PoA existe um controle estrito sobre o participante, que pode, inclusive, conter passos de registro formal com a apresentação de documentos de identificação, tal como uma habilitação formal em algum processo governamental.

Essa estrutura acaba por minar a capacidade de descentralização e publicidade da rede sendo, portanto, uma opção interessante para o uso em blockchains privadas, tais como em aplicações organizacionais, mas com menor aplicabilidade em redes com grande número de participantes, como uma blockchain pública de uso livre.

2.5 Contas, assinaturas e endereços

A principal função de um bloco dentro de uma blockchain é a de registrar as transações desejadas, incluindo-as à cadeia, de forma segura e imutável. Aqui, recordamos a alusão feita ao Livro Razão contábil, em que transações são incluídas de forma sequencial e que resumem, de forma cronológica, todas as entradas e saídas obtidas para determinada conta. O mesmo é feito com as transações em uma blockchain.

Uma transação normalmente indica uma transferência de titularidade de determinado valor entre contas. Imagine uma

transferência bancária em que determinado valor sai da conta A para a conta B; é exatamente o caso da transação em blockchain. Aqui, trabalhamos com contas que utilizam assinaturas privadas como forma de segurança e que estão localizadas em endereços lógicos públicos e conhecidos.

A seguir detalharemos cada um destes componentes.

Contas

Uma "conta" é uma entidade dentro da rede blockchain que possui determinado valor financeiro em criptomoedas (lembrando que este valor pode ser igual a zero). Ela pode ser uma conta pertencente a um usuário, por exemplo, tal como uma conta bancária, ou pode existir na forma de um *smart contract*, em blockchains que prevejam a existência desta possibilidade, que subsiste no ecossistema da cadeia e cuja existência/execução demandam a transferência de valores financeiros.

Independentemente de seu tipo, contas podem transacionar entre si, porém com algumas diferenças e restrições específicas de cada blockchain, tais como a possibilidade de iniciar transações, custos de criação e suas capacidades.

Uma conta blockchain associada a um usuário, também chamada de EOA (do inglês, *externally owned account*) é, internamente, composta por um par de chaves criptográficas: a chave pública e a chave privada, que servem como uma assinatura segura. Os nomes são explicativos e indicam uma chave de segurança que pode ser de conhecimento público, sem riscos imediatos para o titular — a chave pública —, e uma chave que deve ser preservada sob estrita segurança e conhecida somente pelo responsável pela conta — a chave privada. A definição das "assinaturas" será dada mais a frente.

Ao mencionarmos EOAs devemos levar em conta que seus titulares (isto é, usuários da rede) não possuem efetivamente os valores em

criptomoeda de forma direta. A interpretação que deve ser dada aqui é a de que determinados valores tenham sido associados, em algum momento, às suas chaves privadas, dentro do *ledger* da blockchain, e que estes foram enviados para dentro de um ou mais blocos da cadeia através de transações.

O segundo tipo de conta, aquele associado aos *smart contracts*, é identificado não pela chave privada, mas pela própria lógica programada nele e disposta na rede blockchain — no caso da Rede Ethereum, existindo na EVM. Maiores detalhes sobre os *smart contracts* serão dados ao longo do livro, a partir do capítulo 4, sobre a anatomia das aplicações Ethereum.

Assinaturas

A chave privada representa o grau máximo de identificação e associação de valores dentro da blockchain, o que faz com que todas as transações nela sejam anônimas; é por isso chamada também de "assinatura". Por ser (o único) conhecedor de seu segredo, o titular da conta tem acesso a realizar movimentações nestes valores associados, e a chave secreta garante ter sido ele o iniciador do processo.

A chave privada do usuário é única e não pode ser recuperada por qualquer meio. Ela deve ser armazenada de forma segura, evitando ser descoberta, mas também não pode ser perdida. Ao perder o acesso à sua chave privada, o titular efetivamente perde sua conta e quaisquer valores que estiverem associados a ela.

Um caso curioso (e, no mínimo, trágico) a respeito do mencionado anteriormente é o do britânico James Howells que, em 2013, se desfez de um HD em que armazenava a chave privada de uma conta associada a 7.500 bitcoins. Considerando a cotação da moeda em 2021 que, em um momento de baixa, bateu os 30 mil dólares, isso equivaleria a aproximadamente 1 bilhão de reais em um cenário econômico pessimista! Há anos Howells vem tentando

realizar buscas no lixão da localidade porém, até o momento, sem sucesso.

Enquanto este grau de segurança seja muito desejado, há de se pesar que seu funcionamento abre brechas para a prática de crimes, tais como a lavagem de dinheiro e o envio de valores para organizações criminosas/terroristas, de forma que não pode ser interceptada por forças de segurança ou por governos, o que gera debates importantes.

Ao receber uma transação assinada, o destinatário, possuidor da chave pública do titular, verifica a origem da movimentação através de mecanismos de segurança e valida a transferência, finalizando o processo.

Neste meio, encontramos facilitadores do processo de armazenamento e utilização das chaves de segurança. As “carteiras” são sistemas eletrônicos (e aqui somos agnósticos à plataforma utilizada) que controlam o acesso e a movimentação das contas. São sistemas que possuem acesso ao par de chaves criptográficas, executando transações em nome do titular e, portanto, seus provedores precisam ser selecionados com extrema cautela.

Endereços

As contas em uma blockchain possuem um "endereço" lógico ao qual as transações podem ser enviadas; este endereço serve como um identificador seguro, que pode ser de domínio público, e é normalmente gerado a partir de sua chave pública, por meio criptográfico, existindo como uma barreira de segurança a mais.

Esse mecanismo de uso de chaves derivadas das chaves criptográficas evita que elas precisem ser expostas na rede, sendo as transações realizadas publicamente a partir de um endereço, com destino a outro endereço e não mais entre identificadores explícitos.

2.6 Transações

Mencionar que as transações representam movimentações de valores entre endereços, neste ponto, deveria remeter automaticamente à ideia da transferência de valores entre contas ou, em um aspecto mais técnico, às associações de valores no *ledger* da blockchain a identificadores únicos.

Uma transação sempre é iniciada por uma EOA, justamente pelo fato de um *smart contract* não possuir uma chave privada associada, podendo no entanto responder a uma transação iniciada e realizar novas transações a partir de então. Sua estrutura é composta basicamente pelo endereço de origem, pelo endereço de destino e pelo valor sendo transferido.

São dois os principais modelos utilizados para representar transações, dependendo da blockchain em uso: o modelo UTXO (*Unspent Transaction Outputs*) utilizado, por exemplo, pelo Bitcoin, e o modelo baseado em contabilidade, utilizado pela Rede Ethereum.

Dentro do modelo UTXO não existe a possibilidade de “modificação” dos valores dentro do ledger por meio de operações matemáticas. O valor total de criptomoedas disponíveis para serem transacionadas deve equivaler, necessariamente, à soma de todas as transações que associam valores a um mesmo endereço. E essas mesmas transações precisam ser utilizadas para gerar novas transações, em sua totalidade.

Por exemplo: um endereço A recebeu, em algum momento, uma transação com 10 criptomoedas. Uma semana depois, recebeu uma segunda transação com outras 15 criptomoedas. O valor total disponível para o endereço A é de 25 criptomoedas.

Caso o titular da conta desejasse fazer uma transferência de 10 criptomoedas para um endereço B, ele poderia enviar diretamente as 10 criptomoedas da primeira transação para B sem maiores

problemas, visto que existiu uma transação de 10 criptomoedas no passado, com destino em A.

No entanto, caso ele desejasse fazer uma transferência de 20 criptomoedas, ele necessariamente deveria selecionar utilizar as 10 criptomoedas da primeira transação, as 15 criptomoedas da segunda transação, então realizar uma transação de 20 criptomoedas para B e aí realizar uma segunda transação, de 5 criptomoedas, para si mesmo (A enviando para A) como se fosse o troco. Todas as transações envolvidas precisariam ser enviadas para blocos na cadeia e o bloco deveria ser minerado ou validado e aceito como correto, para que as transações fossem finalizadas.

No modelo baseado em contabilidade, esta abordagem é um pouco mais próxima do que nós esperaríamos. Considerando o mesmo exemplo do endereço A, possuidor de 25 criptomoedas, que deseja enviar 20 criptomoedas para B, o modelo baseado em contabilidade criaria uma transação subtraindo 20 criptomoedas do total de A, e uma segunda transação adicionando 20 criptomoedas ao total de B, de forma direta.

Ambos os modelos possuem vantagens e desvantagens. Uma grande vantagem do UTXO diz respeito à sua maior escalabilidade, uma vez que as transações não estão ligadas ao estado da conta — o que é, ao contrário, uma desvantagem do modelo baseado na contabilidade. Por outro lado, a simplicidade do modelo contábil também é uma vantagem em detrimento do UTXO, que é menos intuitivo e que permite seu uso de forma facilitada no desenvolvimento de *smart contracts*.

Quando uma transação é criada, ela é enviada para uma entidade chamada *Mempool* (*Memory pool*, ou "*pool* de memória"), que é uma área na blockchain em que transações ainda não confirmadas ficam em uma espécie de lista de espera, aguardando serem incluídas em um bloco e, finalmente, mineradas. Em uma blockchain, cada participante da rede possui uma cópia do *Mempool*, constantemente atualizada.

Quando um bloco é criado e adicionado à cadeia, isso é feito através do trabalho dos mineradores ou validadores, dependendo do tipo de protocolo de consenso usado pela blockchain; o *Mempool* é consultado para que sejam selecionadas as transações que serão enviadas naquele momento e, uma vez o bloco aprovado e validado, elas são removidas desta área de memória.

Uma das prerrogativas do trabalho dos mineradores/validadores é a de que eles recebem recompensas financeiras pelo trabalho realizado. Da mesma maneira, para que as transações sejam atrativas — e, portanto, aumentar as chances de que elas sejam selecionadas mais rapidamente — costuma-se associar uma espécie de taxa à transação para que ela seja incluída em um bloco. Essa taxa é recebida integralmente pelo ator que está desempenhando o papel de mineração/validação, que é quem seleciona as transações que subirão no bloco. Transações com maiores contribuições certamente chamarão mais a atenção do que transações de valores baixos.

No Bitcoin, que utiliza o modelo UTXO, a taxa da transação equivale a qualquer valor que não tenha sido transacionado para outra conta ou transacionado de volta para o endereço de origem.

Considere o exemplo do endereço A que possui 25 criptomoedas, originárias de duas transações: uma inicial de 10 criptomoedas e uma segunda de 15 criptomoedas. Se, no lugar de enviar as 5 criptomoedas de volta para si, como mencionado no exemplo anteriormente, o titular do endereço A enviasse 4 criptomoedas de volta para a conta de origem, a diferença (a 1 criptomoeda faltante) seria automaticamente considerada como a taxa de transação.

No caso do Ethereum, que utiliza o modelo baseado em contabilidade, e que é de nosso interesse neste livro, existe a figura do ***Gas*** — uma taxa associada às transações que considera o esforço computacional necessário para sua realização, e que será vista futuramente com maiores detalhes.

Um bloco não necessariamente possui um número fixo de transações. Entre suas propriedades, blocos podem ter tamanhos máximos definidos, e os mineradores/validadores têm total liberdade para utilizarem este espaço disponível, desde que respeitados os limites de dados.

Como o intuito do participante da rede é o de maximizar o ganho de dinheiro, espera-se que os novos blocos sempre contenham um número de transações muito próximo aos limites do bloco, porém, isso não é uma regra.

2.7 Forking em projetos blockchain

Um último assunto a ser apresentado antes de falarmos sobre a Rede Ethereum, especificamente, diz respeito às mudanças estruturais a que estão expostas as cadeias blockchain ao longo do tempo. Elas são realizadas através de *forks*. Sua ideia é a mesma utilizada por gerenciadores de versão, em que uma pessoa realiza uma cópia de determinado projeto e, a partir dele, realiza alterações — no caso, sem afetar o projeto original.

Forks normalmente surgem quando os responsáveis por um projeto blockchain (ou a comunidade, no caso de projetos de código aberto) decidem realizar alterações em suas regras e/ou políticas, que podem ou não afetar fisicamente a disposição da cadeia.

Eles são divididos em dois grandes grupos: os *forks* em que as alterações propostas são retrocompatíveis com a cadeia, não exigindo que os nós sejam atualizados para continuarem funcionando, e os *forks* em que as alterações não são retrocompatíveis, e que acabam gerando ramificações na cadeia, normalmente seguindo destinos distintos.

Foi o caso, por exemplo, do Bitcoin. Em 2017, uma proposta de atualização chamada *SegWit* (*Segregated Witness*, "testemunha

segregada") foi colocada em prática buscando maximizar o número de transações que pudessem ser colocadas em um bloco, sem alterar seu tamanho. Isto, através de alterações no processo de assinatura — o que não afetou diretamente os blocos nem os nós preexistentes e, portanto, a cadeia foi atualizada, porém, seguiu inalterada. É o que chamamos de um *soft fork*.

Parte da comunidade Bitcoin que se opôs a essa mudança decidiu realizar alterações na cadeia, aumentando, efetivamente, o tamanho dos blocos, mantendo o processo original de assinatura. Assim, em agosto do mesmo ano, foi aplicado um *fork* não retrocompatível do projeto — chamado *hard fork* — que criou uma ramificação na cadeia original, dando origem ao Bitcoin Cash. É um projeto que compartilha do histórico até agosto de daquele ano, porém, que a partir daí seguiu um curso independente.

Curiosamente, logo depois, em outubro, o projeto Bitcoin sofreu um novo *hard fork*, dando origem ao Bitcoin Gold, devido a uma decisão da comunidade que visou alterações no processo de mineração. Tal como o Bitcoin Cash, ele também compartilha um histórico de cadeia com o Bitcoin e a partir de então passou a existir como uma ramificação independente.

É interessante notar que os valores de criptomoedas existentes em uma cadeia que sofre um *hard fork* são mantidos na cadeia principal, e são criados nas novas ramificações, visto que os projetos compartilham o histórico.

No caso do Bitcoin, em 2017, uma conta que tivesse 100 bitcoins em julho e que não realizou movimentações nem recebeu transferências neste período teria, em outubro, 100 moedas Bitcoin, 100 moedas de Bitcoin Cash e 100 moedas de Bitcoin Gold, visto que as novas cadeias se basearam na principal.

A Rede Ethereum também teve alguns forks aplicados; talvez o de maior impacto tenha sido o que dividiu o projeto Ethereum do Ethereum Classic, o que aconteceu como uma resposta a um

ataque de segurança sofrido na blockchain em 2016. Maiores detalhes sobre isso serão vistos em capítulos posteriores.

2.8 Decisões arquiteturais sobre o uso da blockchain

A blockchain, enquanto tecnologia, pode ser aplicada em diferentes contextos como solução para uma gama de problemas diversos. A opção pela sua adoção, no entanto, deve ser bem estudada, e passar por fases bem definidas, realizadas por um time de arquitetura, como a avaliação de aplicabilidade e o desenho de solução.

Via de regra, a blockchain está assentada em cinco pilares, que são:

1. Imutabilidade.
2. Princípio do não repúdio.
3. Integridade.
4. Transparência.
5. Igualdade de direitos de acesso.

> A "confiança", na blockchain, é estabelecida através das interações entre os participantes da rede.

Um bom projeto de arquitetura que se proponha utilizar a blockchain deve versar sobre temas como a questão do uso da *descentralização* e ponderar assuntos como *responsabilidades e capacidades* dos atores envolvidos. Deve levar em conta questões sobre *vulnerabilidades e a permissão da entrada de novos usuários*, ou mesmo temas como a *criação e publicação da infraestrutura*.

Outros temas de importância, como a *seleção do protocolo de consenso*, também possuem *trade-offs* claros entre *segurança* e

capacidades de escalabilidade, *eficiências de custo* e mesmo *flexibilidade de uso*, por exemplo, e não podem deixar de ser endereçados (e isso deve ficar muito claro para você). Citam-se ainda as questões de *tolerância a falhas*, *velocidade e latência* e até mesmo *privacidade*.

Seu funcionamento é totalmente dependente da existência de atores (pessoas) que desejam fazer parte de sua estrutura, agindo como participantes que permitem que novos blocos sejam incluídos à cadeia constantemente, ao passo que garantem a segurança e a manutenção dos dados na rede. Essa atração é conseguida através do uso de incentivos (normalmente financeiros), porém, um planejamento eficiente de uso de blockchain deve ser capaz também de inibir atores maliciosos de agirem, mesmo sabendo que eles são incentivados a participarem, assim como qualquer outro ator.

Outras questões de segurança a serem levadas em conta envolvem questões de garantia de anonimidade. Aqui, costuma trazer bons resultados a utilização de técnicas como o *mixing* — em que uma transação incluída na blockchain é aparentemente dividida entre diferentes endereços (dando a impressão de ter sido feita por várias pessoas diferentes) — ou de *ring signatures* — em que um "grupo" de usuários compartilha a assinatura sendo, portanto, impossível saber quem foi o usuário dentro do grupo que assinou a transação. A contrapartida é justamente a de que se torna impossível identificar o responsável por determinada ação, o que pode não ser desejável em determinadas situações, em que seja necessária a garantia de origem.

Um ponto importante a ser mencionado e reforçado sempre que possível é que a blockchain deve ser encarada como uma *parte* da aplicação, e não como um sistema inteiro. Assim, ao se criar um projeto de arquitetura, deve-se considerar a possibilidade de utilização de serviços e soluções externas, como bancos de dados comerciais, ERPs, CRMs, aplicações em nuvem, e muitos outros, sempre que necessário e vantajoso.

Um exemplo claro é o gerenciamento de segredos, como senhas: não faz sentido deixar esses dados disponíveis de forma pública, expostos em um bloco com livre acesso. Ainda que devidamente criptografado, é um tipo de dado que não deve ser acessível e que, portanto, é um bom candidato a ser armazenado e manipulado fora da blockchain, embora consumido por ela.

Um trabalho muito interessante sobre a análise de aplicabilidade da blockchain para os processos foi apresentado por Sing Kuang Lo, Xiwei Xu, Yin Kia Chiam e Qinghua Lu em conferência sobre a engenharia de sistemas complexos, da IEEE, em 2017 (confira nas referências bibliográficas). Nele, os autores citam fatores a serem levados em conta quando se discute a opção pelo uso de uma blockchain, como:

1. Multiparidade — a aplicação deve (ou pode) atender a um número diferente de usuários, de áreas diversas, sem restrições? Por exemplo, uma aplicação que seja compartilhada entre fornecedores, clientes e organizações parceiras é uma aplicação com multiparidade, em que não há restrições sobre os dados disponíveis;

2. Autoridade confiável — a aplicação precisa de uma única autoridade, centralizada, capaz de validar as transações, ou isso pode ser compartilhado? Por exemplo, processos que dependam de autorizações ou ações por parte de governos, organismos credenciados ou grandes entidades, como bancos, não são candidatos à descentralização;

3. Operação — a aplicação continuará sendo usada sob as mesmas circunstâncias ao longo do tempo? Dados em blockchain são imutáveis e, portanto, alterações estruturais em projetos são muito complexas. Processos que possam sofrer alterações operacionais, como mudanças constantes em regras ou o aparecimento de novos requisitos, precisam ser bem especificados e discutidos, para avaliar sua aplicabilidade à blockchain;

4. Imutabilidade dos dados e não repúdio — a aplicação suportará trabalhar com dados imutáveis, para os quais deve haver uma política de não repúdio? Neste caso, ações como necessidade de alterações cadastrais constantes ou manuseio de conteúdo ilegal ou secreto, por exemplo, são processos que muito provavelmente devam ser feitos externamente;

5. Alta performance — a aplicação exige tempos de latência e resposta extremamente curtos (quase automáticos) ou deve manipular um grande número de dados? Aqui, a opção pelo uso de uma blockchain passa por uma etapa de seleção da plataforma a ser utilizada, visto cada uma delas possuir tempos e possibilidades diferentes;

6. Transparência — a aplicação deve (ou pode) deixar meus dados públicos? Os processos envolvidos devem ser preparados trabalhar com questões de confidencialidade de informações, necessidade de anonimato ou o uso de criptografia para trafegar dados sensíveis;

Por fim, vale ressaltar que o desenho de uma solução arquitetural com uso de blockchain deve levar em conta fatores como tamanho de bloco e frequência de inclusão, protocolo de consenso utilizado, tipo da blockchain (pública, privada, permissionada...), a estrutura do *ledger* utilizado, mecanismo de criptografia aplicado (que impacta questões de performance, transparência e governança), número de participantes, nível de descentralização, disponibilidade dos processos, entre outros.

Blockchains podem ser vistas como grandes repositórios de dados (e aqui levamos em consideração que os blocos adicionados à rede possuem um histórico de transações, seguros e imutáveis) ou como grandes aparatos de infraestrutura, usados inclusive para comunicações máquina-máquina, através da execução de trechos de código consumidos por aplicações externas.

Dois pontos são fundamentais em um desenho arquitetural de solução que utilize a blockchain: o *armazenamento* e a *computação*.

Tanto o armazenamento como a computação são candidatos a serem realizados na própria cadeia da blockchain, ao que chamamos *on-chain*. Isso permite aproveitar os benefícios trazidos pela própria tecnologia, como as questões de segurança envolvidas.

Apesar disso, como já vimos, sua existência ali pode não ser desejável por questões operacionais, de segurança ou mesmo por custos e, muitas vezes, apenas trechos de informação — metadados — é que são armazenados nela, ou manipulados, por meio de *smart contracts*.

Algumas técnicas de trabalho com blockchain envolvem o uso de cadeias auxiliares — isto é, cadeias externas (daí a expressão *off-chain*), não pertencentes à blockchain, porém, que podem compartilhar com ela algumas ações e funcionalidades.

Os *trade-offs* aqui novamente incluem questões de segurança e escalabilidade, considerando que estamos garantindo a descentralização. Estas técnicas incluem (mas não se limitam a):

1. Uso de cadeias laterais, que diminuem a demanda computacional da cadeia principal.
2. Uso de blockchains privadas, em conjunto, que podem ficar responsáveis por áreas mais críticas da aplicação.
3. Uso de miniblockchains (blockchains temporárias), restritas à execução de determinada tarefa.

Além de funcionalidades, o poder computacional de diferentes cadeias também pode ser compartilhado.

Aqui, falamos de técnicas como o *merged mining*, que utiliza o poder computacional de uma blockchain maior por compartilhar com

ela, simultaneamente, o algoritmo de consenso, ou o *hooking*, em que a execução de suas ações é feita na infraestrutura de outra blockchain.

Não entraremos em detalhes de implementações destes padrões, visto ser este um tema a ser discutido conforme o contexto da solução proposta, porém, há de se ficar claro que esse é um trabalho de análise que deve ser feito com extremo rigor.

> Grande parte do valor de profissionais blockchain está em conhecer as diferentes nuances de propriedades e capacidades disponíveis, entre as diversas opções de cadeia, para a tomada da decisão considerada ótima para o contexto em questão.

Conclusão

Nos tópicos anteriores apresentamos o funcionamento genérico de uma blockchain. Enquanto existem diferenças entre diferentes projetos de cadeias de blocos, seus fundamentos de funcionamento são os mesmos e, por isso, a importância de seu detalhamento.

Finalizamos o capítulo levantando algumas questões importantes sobre os projetos de arquitetura de sistemas que se proponham a usar a blockchain. A estrutura descentralizada da cadeia traz ganhos bastante importantes, porém, por suas próprias características, as aplicações que desejem se aproveitar dela devem ser muito bem planejadas para se obter o máximo de ganho.

A partir de agora nosso foco passará exclusivamente para o Ethereum, e suas particularidades em relação ao modelo geral da blockchain serão apresentadas nos próximos capítulos, que terão como base de entendimento o que foi exposto até aqui.

Capítulo 3
A Blockchain Ethereum

3.1 Introdução

A partir deste capítulo nos concentraremos exclusivamente na discussão do Ethereum, uma blockchain pública não permissionada, de uso geral, com código aberto e conhecido. Esses fatores permitem o desenvolvimento de diferentes *clients* que possibilitam, a um usuário comum, a interação direta com a Rede. Como diferencial, seu protocolo (acessível oficialmente também por diversas linguagens de programação, com *frameworks* específicos) permite a execução de trechos de códigos, através dos chamados *smart contracts*.

Idealizada e criada por Vitalik Buterin no início da década de 2010, o projeto do Ethereum foi oficialmente trazido a público no final do ano de 2013. Sua oferta inicial de moedas (ICO, ou *Initial Coin Offering*) — isto é, a apresentação formal da ideia, etapa em que o projeto é apresentado e aberto a investidores, que levantam os primeiros valores em criptomoedas para serem utilizados na rede — ocorreu em julho de 2014, tendo recebido cerca de 17 milhões de dólares no total para a criação de sua estrutura.

A blockchain Ethereum foi lançada oficialmente em 2015, tendo o bloco gênese sido minerado em julho daquele ano.

Uma das primeiras aplicações criadas no Ethereum, com grande sucesso, foi o jogo *CryptoKitties*. Nela, usuários adquiriam e cuidavam de gatos virtuais (tal como um *Tamagochi* dos anos 1990), porém, diferentemente do bichinho virtual original, um *CryptoKitty* tinha a possibilidade de cruzar com outro gato virtual e gerar sua prole.

Através da execução de algoritmos genéticos, os filhotes gerados pelo cruzamento dos dois *Kitties* apresentavam características (genotípicas e fenotípicas) de ambos os pais, possibilitando a geração de um grande número de variações de gatos, que podiam ser transacionados entre os participantes. O jogo teve um sucesso tão impressionante que a Rede Ethereum chegou a ficar congestionada em seu princípio, impedindo seu uso para outros fins.

Hoje a Rede Ethereum é responsável por aplicações em uma série de domínios, como a indústria da moda, saúde e financeiro, apenas para citar algumas.

Mais recentemente, a popularização dos mecanismos de DeFi e dos NFTs estimularam ainda mais sua utilização como blockchain de desenvolvimento. DeFi é a sigla para Finança Descentralizada (*Decentralized Finance*), e indica uma série de mecanismos que podem ser criados e executados na blockchain do Ethereum, permitindo a oferta de serviços financeiros sem a presença de uma figura centralizadora, como um banco. NFTs (sigla para *Non-Fungible Tokens*), por sua vez, promovem garantia de origem e autenticidade a arquivos, e são muito usados na manutenção e transação de direitos autorais.

Os fundamentos de uma blockchain foram dados com detalhes anteriormente, porém, de forma genérica, e servem para uma variedade de projetos de cadeia de blocos. Nas próximas seções apresentaremos as particularidades do projeto Ethereum e discutiremos as principais características que dotam à Rede de um poder de processamento muito grande, sendo a principal blockchain pública utilizada, hoje, no desenvolvimento de aplicações.

3.2 Blocos no Ethereum

A blockchain do Ethereum pode ser definida tecnicamente como uma "máquina de estados baseada em transações" o que, em computação, representa um sistema cujo estado é alterado de acordo com as transações recebidas (ou as operações executadas).

De acordo com este modelo, o estado inicial da rede — chamado "estado gênese" — representa o estado em que nenhuma interação tenha ocorrido na rede. A partir dele, e de acordo com o número de transações ocorridas desde então, chegamos ao estado atual, chamado também de “estado final”, definido pelo resultado do acúmulo de todas as transações ocorridas na rede a partir do estado gênese.

Assim como no modelo clássico de uma blockchain, no Ethereum, as transações são adicionadas à Rede Ethereum por meio de “blocos”; estes, minerados ou validados (a partir do Ethereum 2.0), adicionados à cadeia, e mantidos ali através das validações dos *hashes* criptográficos do bloco anterior.

Um bloco no Ethereum é composto por um cabeçalho e por uma área reservada às informações de transações. No cabeçalho, algumas informações são comuns ao que foi apresentado previamente, no capítulo introdutório à blockchain; outras são específicas desta blockchain. Os principais dados que compõem um bloco são:

1. *blockNumber* (número do bloco) — indica o número do bloco em questão. Este número é incrementado em 1 em relação ao bloco anterior, sempre que um novo bloco é adicionado à cadeia.

2. *timestamp* (data e hora) — indica a data e a hora em que o bloco foi adicionado à rede.

3. *transactions* (transações) — indica o número de transações incluídas no bloco.

4. *minedBy* (endereço do minerador/validador) — indica o endereço lógico do responsável pela mineração/validação do bloco.

5. *blockReward* (recompensa do bloco) — representa a recompensa recebida pelo minerador/validador responsável pela inclusão do bloco à cadeia.

6. *difficulty* (dificuldade do bloco) — representa o grau de dificuldade considerado no processo de mineração; é representativo do esforço empenhado e dá uma ideia de quantas tentativas de geração de *hashes* precisarão ser feitas até que o desafio lógico seja solucionado. Este valor é ajustado dinamicamente de acordo com o número de agentes, e busca manter um tempo médio de cadência de geração de blocos estável ao longo do tempo.

7. *totalDifficulty* (dificuldade total) — representa a dificuldade total da cadeia, até a adição do bloco em questão. É um valor que acumula historicamente.

8. *size* (tamanho) — representa um indicativo aproximado do tamanho do bloco (em bytes).

9. *gasUsed* (*Gas* utilizado) — representa o total de *Gas* utilizado na totalidade do bloco.

10. *gasLimit* (limite de *Gas*) — indica o limite total de *Gas* recebido por todas as transações do bloco.

11. *baseFeePerGas* (taxa-base por *Gas*) — parâmetro utilizado a partir da atualização *London* do Ethereum, indica a taxa-base, mínima, cobrada em *Gas*, considerada para o bloco em questão.

12. *burntFees* (taxas "queimadas") — parâmetro utilizado a partir da atualização *London* do Ethereum, indica a quantidade de Ether "queimada" após a inclusão do bloco.

13. *MixHash* — o *hash* criptográfico que representa o bloco atual, considerando o cabeçalho.
14. *parentHash* (*hash*-pai) — representa o *hash* do bloco imediatamente anterior. Ele é usado como controle para garantir a imutabilidade dos dados da cadeia ao longo do tempo, protegendo o sistema contra alterações.
15. *Nonce* — representa o resultado do desafio lógico solucionado pelos mineradores. É um *hash* que, quando combinado com o *mixHash*, provê um resultado aceitável para o desafio lógico apresentado para o bloco.
16. Lista de transações — a lista de todas as transações incluídas no bloco.

Um cabeçalho de um bloco contém três estruturas de árvores, além dos dados de identificação, dedicadas a armazenarem metadados sobre o sistema e identificadas pelo seu *hash*. São elas:

1. *stateRoot* — estruturas que guardam informação a respeito do bloco.
2. *transactionsRoot* — estruturas que guardam informação a respeito das transações incluídas no bloco.
3. *receiptsRoot* — estruturas que guardam informação sobre os comprovantes de cada uma das transações incluídas no bloco.

Essas estruturas recebem o nome de Árvores Merkle-Patrícia, uma estrutura que combina as propriedades da Árvore Merkle apresentada anteriormente, com um *Trie*, isto é, uma estrutura em árvore organizada por prefixos, que facilita a busca de dados.

Além de serem usadas no cabeçalho dos blocos, elas são muito úteis quando desejamos realizar armazenamento dentro do Ethereum, por permitirem o armazenamento de chaves e valores de forma segura, criptográfica, em diversos tipos de dados.

Adicionalmente, o cabeçalho do bloco pode incluir também alguns *logs*, que armazenam informações sobre as transações e mensagens criadas a partir de eventos durante sua criação. Esses *logs* são armazenados em uma estrutura conhecida como "Filtro de *Bloom*", uma estrutura de dados probabilística criada por Burton Bloom na década de 1970, que valida de forma eficiente se determinado elemento pertence ou não a um conjunto, o que é útil na organização em situações nas quais existe um grande número de dados a serem computados.

Um *Filtro de Bloom* compreende o uso de um vetor de bits de tamanho *m*, todos com valor inicial igual a zero.

Quando um elemento é adicionado ao filtro, ele é submetido a um número *k* de funções de *hashing*, que determina quais posições do vetor devem ficar com o valor 1.

Para verificar se determinado elemento pertence ou não ao conjunto, basta executar as *k* funções de *hashing* contra o elemento a ser consultado, e verificar o valor de cada uma das posições no vetor.

Se pelo menos uma das posições estiver com valor 0, significa que o elemento não pertence ao conjunto.

O fator de dificuldade associado ao bloco afeta diretamente o valor de seu *Nonce*. Matematicamente, os termos podem ser relacionados pela expressão $\mathbf{n=(2^{2}56/Hd)}$, em que *n* representa o *Nonce* e *Hd* representa a dificuldade.

Essa relação é usada no ajuste de dificuldade tendo em vista a manutenção do tempo de adição de um novo bloco à rede, que no Ethereum fica próximo dos 15 segundos.

Um bloco no Ethereum não tem um tamanho fixo especificado, podendo, virtualmente, variar livremente em quantidades de *Gas* de

até 30 milhões de unidades. Isso é controlado através de uma unidade chamada de “limite de *Gas* do bloco” (*block gas limit*), que é determinada dinamicamente pelos mineradores da rede, representando o quanto eles estão dispostos a aceitar usar de recursos para seguir o processo de inclusão dos blocos.

Cada uma das transações candidatas, por sua vez, possui um valor de *Gas* associado dependendo de sua complexidade. Transações são selecionadas e passam a compor o bloco quando a soma dos valores do seu *Gas* individual é menor ou igual ao limite de *Gas* do bloco.

O *Gas* é uma unidade que mede a quantidade de esforço computacional requerida para a execução de qualquer tipo de ação dentro da Rede Ethereum. Ele será visto com maiores detalhes mais à frente.

> O grande benefício de se trabalhar com tamanhos ajustáveis de bloco é que a própria rede se autogerencia, equilibrando a demanda com as capacidades.

Visto que diversos nós participam da rede e que, em um processo de mineração, os agentes concorrem para encontrarem a solução do desafio lógico, é factível a ocorrência do evento em que dois blocos sejam adicionados a cadeias diferentes de forma simultânea antes que elas sincronizem com o restante da blockchain. Caso este fato não seja bem controlado, ele terminará por gerar ramificações indesejadas na cadeia.

De forma a lidar com este tipo de situação, o Ethereum implementa um protocolo chamado *GHOST* (*Greedy Heaviest Observed Subtree* — em português, algo próximo a “subárvore gananciosa mais pesadamente observada”, ao pé da letra). Em termos gerais, a aplicação do *GHOST* indica que, no caso da ocorrência de algum tipo de conflito entre cadeias, deve-se optar pela cadeia que foi mais

processada (ou computada); indiretamente, falamos aqui de selecionar a cadeia com maior número de blocos.

Um bloco válido que tenha sido adicionado a uma cadeia, porém, que tenha sido rejeitado pelo *GHOST*, não perde a sua validade. Ele passa a existir como um bloco órfão, ao qual não podem ser adicionados futuros blocos. Os mineradores responsáveis pela sua inclusão recebem uma recompensa pelo seu trabalho, mas em valor inferior em comparação com um bloco comum, adicionado à cadeia.

Esses blocos normalmente são referenciados pela expressão "*ommer*" ou mesmo por blocos-tios (*uncle blocks*), por compartilharem o mesmo bloco pai que o bloco aprovado pelo *GHOST*.

A ocorrência de blocos *ommer* é indesejada dentro da cadeia por representar algum tipo de comportamento anômalo da rede e, justamente por isto, eles são utilizados como indicadores no monitoramento da cadeia.

A taxa de blocos-tios (*uncle rate*) é definida pela quantidade de blocos *ommer* sendo formados ao longo do tempo. Valores de limite de *Gas* do bloco muito elevados causam uma lentidão no processo de inclusão de novos blocos, visto demandar maiores poderes computacionais. Como consequência, a probabilidade de formação de blocos *ommer* é aumentada.

Ao contrário, quando os valores de limite de *Gas* são muito baixos, novos blocos são adicionados constantemente, diminuindo a probabilidade da geração de *ommers*. Essa situação, no entanto, apesar de parecer ser positiva, pode causar congestionamentos na rede.

Assim, procura-se manter um valor ótimo, estável, de geração de *ommers*, que indica um valor justo determinado para o limite de *Gas* do bloco, evitando as situações extremas que podem acontecer na blockchain, medidas através do *uncle rate*.

3.3 Ether

A criptomoeda oficialmente associada à blockchain Ethereum é chamada Ether, abreviada por ETH e simbolizada pela letra grega Xi (?). Sua menor subdivisão é chamada wei, sendo que um ETH equivale a 1×10¹8 weis. Esta subdivisão é a unidade utilizada para todas as transações realizadas na blockchain, por padrão.

O ETH é denominado, pelo projeto Ethereum, como sendo “o dinheiro digital global”, ou ainda, “a moeda das aplicações Ethereum”.

O Ether sustenta o sistema financeiro do Ethereum, ainda que não seja a *única* divisa financeira utilizada aí (isso ficará mais claro quando falarmos de *tokens*). Ele possui uma política monetária pública que pode ser consultada livremente a partir da documentação oficial, e que é atualizada constantemente.

O quadro a seguir apresenta os principais múltiplos do wei utilizados no dia a dia, junto com seus nomes comuns.

Nome oficial	Nomes comuns	Conversão
Wei	Wei	1×10° wei
Quilowei / kwei	Babbage	1×10³ wei
Megawei / Mwei	Lovelace	1×106 wei
Gigawei / Gwei	Shannon	1×10? wei
Terawei / Twei	Szabo / Microether	1×10¹² wei
Petawei / Pwei	Finney / Miliether	1×10¹5 wei
Ether	Ether / Buterin	1×10¹8 wei
Quiloether	Grand	1×10²¹ wei
Megaether	Megaether	1×10²4 wei

O ETH é utilizado como o meio de transação na Rede Ethereum, sendo usado para pagar taxas cobradas por meio do *Gas* (descrito com detalhes na próxima seção), serve como moeda em aplicações de finanças descentralizadas, podendo ser usado para a realização de empréstimos, por exemplo, e conta com alto poder especulativo, sendo muito utilizada em carteiras de investimento.

O mercado das criptomoedas em si vem ganhando muito interesse nos últimos anos, impulsionado, principalmente, pelo Bitcoin. Devido às suas arquiteturas descentralizadas e suas seguranças criptográficas, que permitem um maior poder de anonimização na rede, as criptomoedas são alvos constantes de críticas, principalmente por meio de governos, devido ao fato de elas não permitirem um controle estrito — o que tem implicações tanto financeiras, pela dificuldade em cobrar impostos e taxas, como de segurança, em caso de financiamento de organizações criminosas ou terroristas. Apesar disso, o mercado de criptomoedas vem crescendo ano a ano, não sendo difícil nomear ativos digitais que tenham tido valorização de mais de 1000 vezes em pouco tempo.

Academicamente, a economia aceita classificar determinado ativo como “moeda” com base em cinco requisitos:

1. Portabilidade;
2. Durabilidade;
3. Divisibilidade;
4. Fungibilidade;
5. História

O valor de uma criptomoeda é dado, assim como para uma moeda oficial, pelo seu poder de troca. Por ser um ativo digital, tanto a portabilidade como a durabilidade estão garantidas devido à sua própria natureza. O ETH é divisível em subunidades, como já visto, possui boa fungibilidade, e um histórico de funcionamento como meio de troca desde 2015.

3.4 Gas

Já introduzimos o conceito do *Gas* ao mencionar seu limite como forma de controle do tamanho dos blocos na rede. Como consequência, o número de transações inseridas no bloco também é variável, visto cada uma possuir um valor de *Gas* próprio, diferente, dependendo da complexidade computacional esperada para ela.

O *Gas*, na Rede Ethereum, serve como uma unidade de medida para as capacidades computacionais e de armazenamento na blockchain. É usado tanto no controle da cadência de crescimento da cadeia como também (principalmente), desempenhando o papel de um mecanismo de segurança na rede que impede a ocorrência de ataques do tipo do DDoS, ou a execução de trechos de códigos em um *loop* infinito, por exemplo.

Uma vez que a execução de qualquer tipo de computação na rede é cobrada, no momento em que a carteira fica sem fundos (como consequência do consumo de taxas devido ao *Gas*), as transações na rede para aquele endereço são interrompidas, impedindo a continuidade do ataque ou da execução em questão.

Até pouco tempo atrás, os valores (financeiros) por unidade de *Gas* eram expressos em Gwei, ou o equivalente a 1×10?? ETH, e denominados "Preço do *Gas*"; hoje, estão em desuso. Os custos de *Gas*, por sua vez, são calculados com base no funcionamento da EVM e são tabelados, baseados na complexidade de suas operações, sendo expressos em "unidades de *Gas*".

Exemplificando: a adição de dois números é uma operação computacionalmente menos complexa que a operação de cálculo de um *hash* criptográfico, logo, aquela possui um custo menor de *Gas*, expresso em unidades, do que esta.

Em números da revisão de 2017, a adição de dois números custava 3 unidades de *Gas* enquanto a consulta do *hash* (calculado

previamente) de um dos últimos 256 blocos estava tabelado com custo de 20 unidades de *Gas*. O custo do cálculo do *hash*, por exemplo, era feito via uma fórmula matemática que levava em conta o número de palavras, tendo valor mínimo de 30 unidades de *Gas*.

Quando falamos de operações de armazenamento, o custo de *Gas* é calculado com base no menor múltiplo de 32 bytes utilizado para armazenar o dado. Considerando que ele é repetido ao longo da rede, esta é uma operação extremamente custosa para a cadeia e, portanto, existe o incentivo a realizar o armazenamento apenas de pequenas informações. Isso é conseguido através dos custos mais elevados de *Gas* para esse tipo de operação.

É importante diferenciar as expressões “custo do *Gas*”, que indica o número de unidades de *Gas* cobradas em cada tipo de operação, e “preço do *Gas*”, este em desuso, expresso em Gweis e dinâmico, representando valores monetários efetivamente.

Até a atualização London, o preço total de uma operação era calculado multiplicando-se o custo do *Gas* por seu preço naquele momento; quando o usuário gerador da transação estava disposto a oferecer mais *Gas* (pagando, portanto, um preço maior), as probabilidades de que suas transações fossem incluídas mais rapidamente a um bloco aumentavam. Hoje, o cálculo é feito multiplicando-se o limite do *Gas* pela somatória da taxa-base definida para o bloco, com uma espécie de "gorjeta", ofertada em Gweis, cujo valor também influencia na velocidade com que uma transação é inserida a um bloco.

Ao consultar a documentação do Ethereum, pode chamar a atenção a existência de algumas operações com um valor negativo para o custo do *Gas* correspondente. Isso é um incentivo da rede para que o ambiente seja mantido “limpo”, e ocorre quando a operação visa liberar espaço, seja de processamento, seja de armazenamento. Nestes casos, um valor de *Gas* é devolvido ao endereço que o solicitou.

Nos últimos anos, devido ao grande sucesso da Rede Ethereum, os preços do *Gas* subiram muito rapidamente como consequência da competição pela inclusão de transações nos blocos sendo adicionados à cadeia. O que, *a priori*, seria um sinal muito positivo para a comunidade desenvolvedora, teve um impacto bastante negativo quando seu alto valor passou a inibir seu uso.

Como forma de atenuar esse efeito, algumas iniciativas foram tomadas. Talvez a mais significativa, chamada de “dimensionamento da camada 2” (*layer 2 scaling*), compreendeu uma série de iniciativas que atuaram justamente no dimensionamento da rede, retirando as transações da camada principal da cadeia, aumentando sua capacidade, e portanto diminuindo os custos de transação. Aqui, algumas soluções comerciais foram implementadas, enquanto outras seguem sendo preparadas.

O Ethereum 2.0 (chamado também de Eth2 no passado, hoje "camada de consenso"), que será visto mais à frente, realizará uma atualização na rede que implementará funcionalidades de *sharding*, que diminuirão os impactos de uso da rede, e espera-se que consiga fazer a migração do protocolo de consenso (do PoW para o PoS) por completo, diminuindo os custos associados às etapas de mineração (que passarão a ser feitas por validação). Esse projeto é muito aguardado e deve ser implementado por completo em um futuro muito próximo.

Quando uma transação é enviada para a rede, o usuário define uma variável chamada de limite de *Gas* (*Gas limit*), indicando o valor máximo de *Gas* que ele está disposto a pagar. Quando essa transação é enviada para o bloco minerado, a EVM verifica o custo (em *Gas*) necessário para a operação. Caso o limite de *Gas* definido seja superior ao necessário, a operação é concluída e qualquer unidade de *Gas* que tenha sobrado é devolvida ao usuário diretamente. O minerador recebe, então, uma gorjeta (ofertada por quem iniciou a transação) como um bônus por seu trabalho.

Caso o limite de *Gas* estabelecido seja menor que o custo necessário para a operação, ela não é concluída, sendo lançada uma exceção do tipo *Out of Gas Exception* ("sem *Gas*"), e os valores incluídos não são devolvidos.

3.5 Transações no Ethereum

Anteriormente definimos a Rede Ethereum como uma máquina de estados. Sendo isso verdade, a “transação” é quem desempenha o papel de realizar as mudanças de um estado a outro. Não é demais relembrar, ainda, que o estado atual da rede é o resultado do acúmulo de todas as mudanças de estado — estas, consequência das mudanças geradas por cada uma das transações incluídas à blockchain.

Uma transação Ethereum é tecnicamente definida como uma “mensagem binária serializada, assinada criptograficamente”. Inicializada por uma EOA (uma conta pertencente a uma pessoa), ela é incluída à Rede Ethereum através de blocos.

Sua serialização é feita nativamente através de um mecanismo de codificação chamado RLP (*Recursive Length Prefix*, ou “prefixo de comprimento recursivo”). Esse modelo agrupa os dados em dois tipos: *Strings* (entendidas como sequências de bytes) e *arrays* (entendidos como listas), em que uma *String* é um tipo “simples”, enquanto um item do *array* pode ser ele mesmo categorizado como uma *String* ou como um *array* — e daí o nome “recursivo” para o mecanismo.

O RLP foi criado pelo time de desenvolvimento do Ethereum tanto para serializar o armazenamento de dados na rede como para serializar mensagens de comunicações. É um modelo totalmente consistente, que o permite ser usado para codificar qualquer tipo de objeto, e que utiliza uma notação de prefixos indicando o

comprimento do dado que está sendo serializado, de forma individual. Como resultado, a serialização é feita na forma de números inteiros como *Big-Endians*, sendo múltiplos de 8 bits.

Com a atualização London, o RLP deixou de ser a única forma de serialização das transações. Pela implementação de uma melhoria na rede, novos formatos (não nativos) passaram a funcionar, sendo tratados por meio de mecanismos de "envelope", que permitem que o dado serializado seja incluído à cadeia.

Quando falamos de “transação”, referimo-nos tanto ao envio de mensagens — por exemplo, na transferência de valores de Ether entre uma conta A e outra conta B — como à criação de contratos.

> Contratos não são capazes de iniciar uma transação, mas podem responder a elas e, uma vez ativados, podem, mesmo, chamar outros contratos — o que são chamadas de transações internas.

Sua parametrização básica é composta pela seguinte estrutura:

1. *Nonce* — indica o número de transações enviadas pelo autor, e é um mecanismo de segurança extremamente eficiente. Não deve ser confundido com o *Nonce* do desafio lógico. Mais à frente dedicaremos alguns parágrafos a comentar sobre sua importância.

2. Limite de *Gas* — indica a quantidade máxima de *Gas* que o usuário está disposto a ofertar para executar a transação. Lembre-se de que esta quantidade deve ser superior ao realmente necessário para que a transação seja executada com sucesso.

3. Valor máximo de gorjeta ofertada — indica o valor máximo de *Gas* que o usuário está disposto a ofertar para o minerador/validador do bloco como uma gorjeta por seu

trabalho. Quanto maior este valor, maior a probabilidade de a transação ser escolhida para entrar em um bloco.

4. Recipiente — campo indicativo do endereço lógico do destinatário da transação. Pode ser enviado em branco caso este não exista (por exemplo, na criação de um contrato).

5. Valor — o valor, em Wei, sendo transferido para o destinatário.

Antes da atualização London, o preço do *Gas* era também informado, e indicava o preço de *Gas*, em Wei, que o usuário estava disposto a pagar por unidade de *Gas* necessária para a execução da transação. Como vimos, quanto maior este valor, maior a probabilidade de a transação ser incluída à cadeia mais rapidamente e o produto da multiplicação do preço de *Gas* pela quantidade de *Gas* necessária é que determinava o preço efetivamente pago pelo usuário. Apesar de estar em desuso, blocos antigos do Ethereum ainda possuem este dado armazenado.

Um campo extremamente importante, que não foi mencionado, é chamado "*v,r,s*" e inclui informações utilizadas para a assinatura da transação através do modelo de Assinatura Digital de Curvas Elípticas, que será visto em detalhes ainda neste capítulo ao abordarmos os modos de criptografia na Rede Ethereum. No momento, tenha em mente que toda transação deve ser assinada criptograficamente, como forma de garantia da origem dos dados.

Transações que enviam mensagens podem incluir um campo opcional de dados, que servem para enviar parâmetros, por exemplo, quando o destinatário é um contrato. Já as transações que criam contratos possuem um campo chamado *init*, que faz a inicialização do código — ocorrendo uma única vez, portanto.

Sempre que uma transação é executada na rede, o Ethereum armazena seu comprovante de transação no cabeçalho bloco responsável, que inclui informações detalhadas sobre o uso de *Gas*, apresenta os *hashes* da transação, dá as informações do bloco e também armazena, ali, os *logs* gerados durante sua execução.

Ao mencionar a estrutura de uma transação, citamos a existência de um campo dedicado a armazenar o *Nonce* de transação. Este, segundo Andreas Antonopoulos — um dos atores de mais renome no universo blockchain — é “talvez, o mais importante, e o mais desconhecido componente de uma transação”.

Definido em poucas palavras na documentação oficial do Ethereum como “um número escalar que equivale ao número de transações enviadas por um endereço” ou “o número de criações de contratos feitas por uma conta”, o *Nonce* de transação é um importante atributo de segurança e está ligado ao endereço de origem, não diretamente à blockchain.

A primeira função desempenhada pelo *Nonce* é a de manter a ordem das transações, impedindo, por exemplo, que transações incluídas posteriormente sejam mineradas antes que as transações incluídas em primeiro lugar, o que poderia causar problemas em alguns casos.

Seu segundo aspecto diz respeito a impedir que transações que já tenham sido executadas voltem a ser mineradas por erro, ou de forma maliciosa — o chamado *replay attack* (ataque de repetição), em que a informação pública disponível no bloco é “copiada” e enviada novamente para a rede. Ao manter um registro sequencial do número de transações confirmadas, este tipo de situação é coibida.

A existência do *Nonce* traz alguns desafios importantes para o desenvolvimento de transações. Em primeiro lugar, surge a necessidade de se manter um registro estrito dos valores de *Nonce* já utilizados. *Nonces* repetidos impedem a transação de ser minerada, enquanto *Nonces* que não sejam sequenciais (quando seu valor salta um número, deixando um hiato) também não serão executados até que o número faltante seja enviado para a rede (e este será executado antes).

Esse pode ser um tema extremamente sensível, por exemplo, se considerarmos casos em que transações são geradas de forma concorrente, de forma paralela ou mesmo assíncrona, sem que haja uma fonte capaz de informar o número do *Nonce* a ser utilizado pela transação de forma inequívoca.

Quando uma transação é iniciada por um nó na rede ela é validada e, caso aprovada, é enviada para seus nós vizinhos. Estes a validam novamente e enviam para seus outros vizinhos, processo que é repetido até que todos os nós tenham uma cópia da transação. Somente então é que ela é minerada e incluída em um bloco.

Transações de mensagens enviadas para outro EOA causam uma mudança no estado da rede — a partir do qual a transação é contabilizada. No caso de mensagens sendo enviadas para uma conta de contrato, estas são processadas por meio da EVM e apenas após terem sido executadas corretamente na máquina virtual é que causam a mudança de estado na cadeia.

3.6 Criptografia no Ethereum

A criptografia tem papel de destaque na Rede Ethereum, e é responsável por manter a segurança em uma série de aspectos, aparecendo como peça-chave tanto na manutenção das assinaturas das transações, como garantindo a segurança das contas e, ainda, aparecendo durante a formação dos *hashes*, a partir dos quais os blocos e dados são controlados para, por exemplo, garantia de imutabilidade. Ambos os processos utilizam abordagens diferentes de criptografia, que serão expostas com detalhes ao longo dos próximos parágrafos.

Conforme visto anteriormente, uma conta EOA é composta basicamente por um par de chaves: uma pública, e uma privada. A

chave privada é aquela que representa a conta, propriamente dita, sendo usada na associação com as transações e permitindo sua movimentação e, portanto, devendo ser guardada sobre estrito sigilo. Já a chave pública é utilizada como mecanismo de validação, garantindo que a chave privada utilizada seja válida e, assim, confirmando a origem da transação. O modo com que uma chave privada permite o acesso às contas é conseguido através de uma assinatura.

Uma conta que representa um contrato, por sua vez, é assegurada por sua própria codificação, sem envolver mecanismos diretos criptografia. Isso ficará mais claro quando iniciarmos a discussão sobre contratos e aplicações na Rede Ethereum.

O principal mecanismo de criptografia usada no Ethereum para controle de chaves é baseado na chamada Criptografia de Curva Elíptica (ECC, do inglês *Elliptic Curve Cryptography*), e as assinaturas são geradas a partir do Algoritmo de Assinatura Digital de Curva Elíptica (ECDSA, do inglês, *Elliptic Curve Digital Signature Algorithm*), que utiliza o ECC como base de seu funcionamento.

Uma curva elíptica, na matemática, é usualmente obtida a partir de equações cúbicas, em que números reais utilizados como coeficientes determinam sua forma. Curvas elípticas, em criptografia, são formadas sobre corpos finitos, e são utilizadas para, a partir de uma chave conhecida, gerar uma segunda chave através da aplicação de um operador matemático sobre um ponto conhecido. Isso pode ser simplificado a partir da fórmula $K=k*G$, em que K representa a chave pública, k, a chave privada e G, o ponto gerador.

Note que aqui o operador não representa a multiplicação aritmética e, portanto, não há que se falar que seu inverso seja uma operação de divisão.

Este tipo de operação é classificado como de “caminho único”, uma vez que se desconhece uma maneira de, conhecendo-se o ponto

gerador *G* e a chave pública *K*, chegar ao valor da chave privada *k*. Matematicamente, este é conhecido como o "problema do logaritmo discreto de um grupo G".

A curva elíptica utilizada pelo Ethereum é gerada a partir do padrão *secp256k1*, o mesmo utilizado pelo Bitcoin. Sua implementação é computacionalmente eficiente, sendo até 30% mais rápida que outras curvas de amplo uso. Não entraremos em detalhes sobre o funcionamento ou a teoria das curvas elípticas, visto ser este um tema extremamente denso, estudado por matemáticos com profundidade, o que foge totalmente do escopo deste livro.

Basta-nos, neste ponto, entender que graças ao ECC temos um modo eficiente de gerar uma chave pública que possa ser compartilhada sem o risco de que a chave privada seja descoberta, por qualquer meio, mantendo a segurança do sistema.

O algoritmo ECDSA, que utiliza a ECC como base de seu funcionamento, foi proposto inicialmente em 1992 por Scott Vanstone, em trabalho desenvolvido para o NIST (*National Institute of Standards and Technology*, Instituto Nacional de Padrões e Tecnologia), nos Estados Unidos. É utilizado principalmente em métodos de assinatura digital, por exemplo, no reconhecimento de origem de e-mails, identificação de sites na Internet, reconhecimento de assinaturas em documentos digitais ou mesmo em transmissões para a Receita Federal no Brasil, entre outros, e traz ganhos bastante evidentes em comparação com outros métodos de amplo uso, como o DSA ou o RSA.

Entre suas vantagens, citam-se:

1. O uso de chaves menores em comparação com outros algoritmos análogos;
2. Maior eficiência em tempos de resposta;
3. Menor consumo de espaço de armazenamento.

O ECDSA é um algoritmo robusto, reconhecido pelo IEEE e atende às características desejadas de garantia de autenticidade, não

repúdio — isto é, uma vez aplicado, o autor não pode negar sua autoria —, e integridade, uma vez que, caso o conteúdo da mensagem fosse alterado, isso demandaria uma alteração na própria assinatura.

No processo de criptografia assimétrica (a criptografia que trabalha com chaves públicas e privadas) no Ethereum, a chave primária é gerada de forma aleatória, sendo composta por 256 bits de informação (ou 64 caracteres, em notação hexadecimal).

Normalmente ela é armazenada pela carteira de forma segura ou diretamente pelo usuário; é um processo que não está relacionado de forma alguma à Rede Ethereum, podendo ser realizado de forma totalmente independente. A chave pública do Ethereum, por sua vez, é composta por 65 caracteres em notação hexadecimal.

As assinaturas ECDSA são compostas por dois números inteiros, denominados *r* e *s*, computados durante o processo de assinatura representando, efetivamente, sua saída, definida por *(r,s)*.

A definição matemática dos valores de *r* e *s* é dada pelas relações *r=x1 mod n* e *s=k?¹(z+rd?) mod n*, em que *x1* representa a coordenada do ponto calculada para a curva elíptica, *n* representa a ordem do valor do ponto gerador *G*, *k* um valor único associado à mensagem, *z* representa os bits mais à esquerda do *hash* calculado, de tamanho igual a *n*, e *d?*, a chave privada.

Na verificação, realizada através da chave pública, os valores são utilizados para calcular um ponto da curva elíptica e, a partir daí, verificar a validade da informação.

Durante o processo de verificação da validade da assinatura, as relações *r=(k×G)?(mod q)* e *s=k?¹(H(m)+rt)(mod q)* são válidas, em que *k* representa um valor único associado à mensagem (possivelmente aleatório, ainda que não seja recomendado), *G* representa um ponto na curva elíptica de ordem *q*; *H(m)*, o *hash* criptográfico da mensagem e *t*, a chave pública.

Tendo sido apresentados os conceitos de criptografia assimétrica no Ethereum, agora podemos retomar a discussão sobre os parâmetros utilizados na identificação de transações, citados na seção anterior, que incluíam a tripla “*v,r,s*”.

Os valores *r* e *s* são, efetivamente, os valores obtidos pela aplicação do ECDSA, conforme já apresentado. O valor *v*, por sua vez, é utilizado pelo Ethereum como um parâmetro de segurança, que permite obter a chave pública a partir do endereço enviado na transação, evitando trafegar informações sobre chaves ao longo da rede.

O segundo tipo de criptografia encontrado no Ethereum diz respeito às funções geradoras de *hash*, que ocorrem em diversos pontos do fluxo de informação da blockchain, como na manutenção das Árvores Merkle, que garantem a imutabilidade das transações.

O Ethereum trabalha com um tipo de função de *hash* denominada *Keccak-256* (pronunciada “*ket-chak*”), pertencente a uma categoria de gerações de *hashing* denominadas de "funções-esponja". Elas recebem dados de comprimento variável como entrada e apresentam, como saída, dados de comprimento também variável. No caso do *Keccak-256*, tem-se 256 bits de saída, de forma fixa.

Internamente o *Keccak-256* trabalha através da execução de permutações entre 7 conjuntos definidos em seu protocolo, desenhado de forma extremamente segura, sendo resistente a diversos tipos de ataques. Sua especificação formal é dada pela fórmula `n=256:?Keccak[r=1088,c=512]?256` , em que *n* indica o número de bits de saída; *r*, a taxa de bits, que é o número de bits processados pelo algoritmo; e *c*, a capacidade, relacionada ao nível de segurança da função.

Os parâmetros *r* e *c* estão diretamente ligados à aplicação da função-esponja. Uma função-esponja recebe este nome porque é composta por duas fases principais: a absorção e a compressão, tal

como em uma esponja que, quando molhada, absorve um líquido, porém, quando pressionada, o libera.

Simplificando sem maiores detalhes, entende-se o *Keccak* como um mecanismo em que, durante a fase de absorção, a mensagem cujo *hash* deva ser calculado é dividida em blocos de bits com tamanho igual a *r*. Sobre cada um deles são realizadas operações lógicas XOR comparando com o resultado obtido pelo conjunto *r* anterior (ou, todo composto por zeros, no caso do primeiro conjunto) e, sobre o resultado, aplicada uma função especializada.

O valor de *c*, não mencionado até então, está ligado a parâmetros de segurança, tais como a resistência a colisões, e é utilizado dentro da função especializada, modulando sua saída, gerando um novo conjunto *c* a ser utilizado na próxima iteração.

Esse processo é repetido até que todos os blocos tenham sido processados. Na fase de compressão, é aplicada uma nova função especializada sobre o resultado obtido anteriormente e tomados os primeiros bits, obtendo-se uma saída de tamanho *n*, que resulta no *hash* obtido como saída da função.

3.7 Ethash e Casper

Concentremo-nos agora nos protocolos de consenso utilizados pela rede Ethereum durante o processo de mineração. Tal como visto, o Ethereum utiliza atualmente um processo baseado em PoW, havendo uma expectativa muito grande para a transição para o uso do PoS, principalmente após a adoção do Ethereum 2.0. Neste ínterim, dois algoritmos são de grande importância: o *Ethash*, que é uma implementação utilizada no PoW, e o *Casper*, que está sendo preparado para ser utilizado no PoS.

O *Ethash* surge na Rede Ethereum como uma atualização do algoritmo de *Dagger-Hashimoto*, algoritmo reconhecido por utilizar

intensivamente a memória. Não apenas seu uso de memória é intensivo, ele também cresce a cada *epoch* — em média, a cada 30 mil blocos minerados, o que cria importantes desafios para mineradores da Rede Ethereum.

O *Ethash* utiliza um valor denominado *seed*, composto por 32 bytes e calculado a cada novo bloco pelo *hash* do *seed* anterior. Este valor é utilizado internamente para a geração de um pseudo "*cache*" de memória, com 16 MB de dados, a partir dos quais é gerado um conjunto de dados na forma de um grafo acíclico dirigido (DAG, *Directed Acyclic Graph*), com tamanho superior a 3 GB — e crescendo a cada *epoch*.

De forma geral, a mineração no *Ethash* envolve a seleção de um valor aleatório presente no DAG e o cálculo do *hash*, combinando o valor selecionado aleatoriamente no conjunto de dados com o valor calculado para o bloco, considerando todas as transações. Considera-se minerado o bloco quando um valor ótimo é encontrado no DAG, que causa o *hash* ficar abaixo do valor *target*, ajustado conforme a dificuldade.

Ao contrário do processo utilizado no Bitcoin, em que o cálculo do SHA-256 depende, exclusivamente, da velocidade de processamento, o grande desafio do *Ethash* é o de que o DAG deve ser armazenado em memória por completo — já que um valor aleatório precisa ser escolhido daí.

Considera-se minerado o bloco cujo *hash* calculado esteja abaixo do valor especificado pelo *target*, ajustado pela dificuldade de mineração, tal como na abordagem tradicional do PoW.

Devido ao uso intensivo de memória requerido pelo algoritmo, é impossibilitado o uso de placas de circuito integradas de aplicação específica (ASIC, do inglês, *Application Specific Integrated Circuits*), que são placas extremamente eficientes para o cálculo de outros tipos de *hashing*, como o SHA-256, utilizado pelo Bitcoin. Ainda que circuitos sejam construídos com uma quantidade de memória

suficiente, a crescente demanda por memória torna estes equipamentos obsoletos rapidamente. Pelo mesmo motivo, a mineração utilizando a CPU é extremamente ineficiente.

O *Casper* é o segundo algoritmo a ser apresentado, e terá sido totalmente implementado quando o Ethereum fizer sua migração para um protocolo de consenso baseado em participação — o PoS. Atualmente, ele se encontra em desenvolvimento, sendo trabalhado por duas frentes diferentes, em projetos denominados *Friendly Finality Gadget* (*FFG*) e *Correct-by-Construction* (*CBC*), ou "Gadget de finalidade amigável" e "Correto por construção", em traduções literais.

De acordo com a documentação do Ethereum, quando a plataforma migrar para o protocolo PoS qualquer pessoa que detenha pelo menos 32 ETH poderá participar como validador, sendo que um depósito de valor maior acarretará em um maior peso nas decisões. Os participantes — chamado *stakers* — são reunidos em um comitê e, a partir de análises feitas por um protocolo em comum decidem se o bloco é válido ou não, dando seu voto e depositando determinado valor, na forma de uma "aposta" no caso de concordância com a validade. A decisão sobre a validade do bloco é tomada quando dois terços dos validadores entrarem em consenso.

Após tomar a decisão, os *stakers* são bonificados com incentivos financeiros proporcionais ao valor apostado, caso o bloco seja aprovado, ou são punidos, perdendo o valor apostado, caso sua aposta não vá ao encontro dos demais.

A grande ideia do PoS é a de diminuir a dificuldade do processo, tornando-o menos dependente de energia (e, portanto, amigável ecologicamente), porém ao mesmo tempo seguro. Neste caso, assegura-se a lisura do processo através dos valores requeridos para se participar como um *staker*: não se espera que uma pessoa que tenha depositado seu dinheiro aja de má-fé, correndo o risco de perder o valor "apostado". Existe ainda o caso de que, se um bloco malicioso for incluído à rede, isso causará uma desvalorização do

Ether, o que também trará perdas financeiras significativas para o validador que não agir de forma correta.

O *Casper FFG* foi apresentado inicialmente como um protocolo híbrido entre o PoW e o PoS, que deveria ser usado durante o processo de transição entre ambos os modelos. De acordo com o *FFG*, o *Ethash* continuaria funcionando, porém, haveria uma espécie de *checkpoint* periódico. Essa ideia foi abandonada pelo time Ethereum, que decidiu trabalhar com o *FFG* na forma de um protocolo mais puro, voltado totalmente para o PoS.

Atualmente o *Casper FFG* é trabalhado com a ideia de *checkpoints* armazenados em uma estrutura de árvores, em que determinados blocos são selecionados e, de forma hierárquica, são validados pelo comitê responsável.

Um *staker* vota em *checkpoints* pertencentes a diferentes níveis da árvore, formando um *link* entre eles, atestando que ambos apresentam transações válidas. O bloco com hierarquia superior, no *link*, é chamado de origem (*source*), e o bloco mais abaixo é chamado de alvo (*target*). A estrutura do *link* é definida no formato *(S,T)*, em que *T*, necessariamente, deva ser um descendente direto de *S*.

Em seguida, o processo passa por duas fases: a justificação e a finalização. Os *links* são passados para votação e, aqui, existem duas rodadas de votos, aprovados pelos *stakers*.

Quando mais de dois terços dos *stakers* entra em um consenso de que um *link (A,B)* é válido, o bloco *B* passa para o estado “justificado”. Em um segundo momento, quando se analisa um *link (B,C)*, e o consenso decida ser um *link* válido, *C* passa para o estado “justificado”, e *B*, que já estava justificado, passa para o estado “finalizado”.

Os votos são tomados em duas rodadas de consenso, e blocos finalizados são considerados aptos para integrarem a cadeia.

O *Casper CBC*, por sua vez, apresenta metodologia diversa. No lugar de coletar votos ao longo da rede, ele utiliza como princípio a criação de uma “mensagem”, que é acompanhada por “justificativas” dadas por validadores.

Durante uma sessão de validação no *Casper CBC*, blocos são validados durante um determinado espaço de tempo. Neste processo são geradas “mensagens” que incluem justificativas para os blocos anteriormente validados, virtualmente unindo os blocos validados em uma estrutura similar a um DAG.

Graças ao modelo de justificativas, é possível manter um histórico do que foi analisado até então, permitindo punir participantes que tenham validado blocos indevidamente, o que reforça a segurança do método.

Neste ponto surge a aplicação de um protocolo chamado **fork-choice**, que considera o conjunto de blocos para os quais já foram criadas mensagens e define, a partir deles, a chamada “cadeia canônica” — isto é, a cadeia que passa a representar o consenso encontrado entre os validadores.

A ideia do *fork-choice* é a de encontrar a cadeia que apresenta maior poder de imutabilidade. Normalmente isso significaria ser a cadeia mais longa, porém, outros aspectos são levados em conta, como os tempos decorridos entre a inclusão dos blocos e mesmo os ganhos obtidos pelas transações, que estão sendo incluídas nos blocos.

O *Casper FFG* será o primeiro mecanismo de consenso utilizado pela cadeia do Ethereum 2.0, quando ela passar a ser utilizada. Apesar disto, o projeto *Casper CBC* segue ativo e, por trabalhar com métodos mais formais que o *Casper FFG*, é considerado pela comunidade Ethereum como complementar a ele, indicando que um híbrido de ambos possa vir a ser utilizado no futuro.

3.8 Tokens e oracles

Tokens e *oracles* (oráculos) são parte fundamental do universo Ethereum, e são mecanismos que permitem à Rede apresentar vantagens competitivas frente a outras blockchains. *Tokens*, como veremos, são mecanismos que servem como acesso ao consumo de serviços na Rede Ethereum, tal como as fichas eram utilizadas, antigamente, em máquinas eletrônicas, ou ainda, como garantidores de posse de algum bem ou ativo. *Oracles*, por sua vez, são os mecanismos através dos quais são feitas comunicações com o exterior da Rede Ethereum para, por exemplo, a obtenção de dados — tal como os disponíveis de serem consultados via API. Sua utilização não é obrigatória, porém, dota à blockchain de grande poder de versatilidade.

Tokens

O uso de *tokens* não é exclusivo do Ethereum, nem é uma novidade criada com ele. Em linhas gerais, um *token* no Ethereum representa uma interface criada a partir da rede, que dá acesso à utilização de serviços disponibilizados aí ou o título de posse sobre determinado ativo, armazenado de forma digital, sem que, para isso, seja necessária a inicialização de uma nova blockchain ou a criação de clientes especializados. Através dos *tokens*, os usuários conseguem consumir serviços na blockchain, de forma facilitada.

Ao contrário do que se possa imaginar, um *token* não é nada além do que uma implementação, em um *smart contract* de um padrão definido pela comunidade, contendo os dados e funcionalidades mínimas que ele deve possuir / ser capaz de executar. A esses modelos-base, chamam-se "padrões".

Diferentes padrões de *tokens* são utilizados pelo Ethereum. Dois deles merecem atenção especial devido a suas aplicações: o ERC-20 e o ERC-721.

Tokens que seguem o padrão ERC-20 são chamados de “fungíveis”. São *tokens* utilizados para a troca de bens ou serviços específicos, tal como uma unidade monetária, e são, provavelmente, os *tokens* mais utilizados por usuários da Rede Ethereum.

Ao adquirir *tokens* ERC-20 de um serviço A, usuários podem comprar mercadorias oferecidas por A ou consumir seus serviços. Serviços como estes são usados, por exemplo, por casas de apostas esportivas virtuais, em que *tokens* são usados como meio de pagamentos.

Um exemplo real, brasileiro, de uso de *tokens* ERC-20 é o Wibx, que oferta serviços de marketing, engajamento e divulgação de marca. Nessa plataforma, a interação de clientes com a exposição da marca gera recompensas em forma de *tokens* Wibx, usados para a troca de produtos e mesmo por dinheiro.

Outro projeto interessante é o Enjin, utilizado por desenvolvedores de games para permitir a compra e troca de itens utilizados nos jogos, em tempo real, utilizando a moeda Enjin Coin (um *token* do Ethereum).

Por atuarem como uma moeda, *tokens* deste tipo são também usados de forma especulativa, não raramente encontrados como meios de investimento, ofertados por organizações especializadas em criptoativos. Esse tipo de *token*, chamado de “*token* de uso”, permite ao usuário usufruir do serviço, porém, não o dota de qualquer tipo de direito de posse na rede.

Tokens do padrão ERC-721, por sua vez, são ditos infungíveis; são *tokens* que não podem ser transacionados para que os usuários usufruam de seus benefícios. Um tipo de aplicação que tem recebido bastante atenção, ultimamente, e que utiliza esse tipo de *token* é a da preservação de direitos autorais, em obras artísticas e outros artigos colecionáveis que, graças à tecnologia blockchain, podem ser sua autenticidade comprovada de forma inequívoca.

Tokens não fungíveis, por suas peculiaridades, recebem um nome especial em vez de serem chamados pelo seu protocolo: NFT. Maiores detalhes sobre eles serão dados no capítulo sobre *Anatomia das aplicações Ethereum*, em um tópico dedicado.

Ainda que o ERC-20 e o ERC-721 sejam os principais padrões utilizados para geração de *tokens* no Ethereum, eles não são os únicos. Existe uma ampla lista de padrões utilizados para tipos com aplicações específicas. Alguns exemplos são o ERC-1400, utilizado por *tokens* de segurança, que asseguram ativos negociáveis (débitos, ações, garantias…) em transações e o ERC-1643, utilizado para assegurar o armazenamento de documentos.

Tokens podem ser usados, ainda, para a utilização de recursos na rede, garantir a propriedade de investimentos — por exemplo, em ouro —, podem permitir o acesso a determinados locais exclusivos para quem possua *tokens* de determinada aplicação, permissão para votar em decisões estratégicas, entre muitas outras.

Oracles

A segunda matéria de importância desta seção são os *oracles* que, como visto, são aqueles mecanismos através dos quais se constrói um meio de comunicação com o exterior da Rede Ethereum. A própria documentação do Ethereum os descreve como sendo “uma ponte entre a blockchain e o mundo real” ou ainda diz que os *oracles* existem “como APIs que transportam informação externa para dentro dos *smart contracts*”.

A utilização de *oracles* é um tema que exige extrema cautela, uma vez que a segurança das aplicações na rede pode ser impactada, exatamente como ocorreria caso um desenvolvedor optasse por consumir uma API de terceiros, em que não possui controle sobre a veracidade e a qualidade dos dados de resposta.

O caso contrário também é possível: *oracles* podem ser utilizados para transportar informação de dentro da blockchain para o exterior,

o que permite, por exemplo, sua utilização para popular o *front-end* de algum tipo de aplicação que queira consultar um *smart contract*. Isso é extremamente interessante, uma vez que possibilita o desenvolvimento de aplicações descentralizadas totalmente funcionais.

Oracles trabalham com três tipos de interação: *request-response*, *immediate-read* e *publish-subscribe*.

Os oráculos do tipo *request-response* têm atuação semelhante às requisições realizadas em uma arquitetura cliente-servidor. Consideram-se aqui as requisições que são intensivas em dados, em que há necessidade de se fazer consultas a sistemas externos, ou mesmo a dados processados, por exemplo, por sensores e dispositivos de IoT.

Oracles de leitura imediata (*immediate-read*) são os mais simples para serem trabalhados, e retornam dados utilizados para uso, sem necessidade de processamento. É o caso de uma consulta que retorne a temperatura em determinada cidade naquele momento. Não há muito a ser discutido aqui.

Já os oráculos do tipo *publish-subscribe* se diferenciam dos de leitura imediata justamente por realizarem uma subscrição ao serviço, em que dados são transferidos periodicamente, sempre que houver algum tipo de alteração. Esses oráculos são úteis, por exemplo, em cenários de acompanhamento de dados em tempo real, como dados referentes ao mercado financeiro ou ao trânsito.

O uso de oráculos não se restringe a consultas. Há oracles especializados em serviços de computação e processamento, sendo oferecidos, inclusive, por provedores em nuvem.

Assim como um *token*, um *oracle* também se baseia na implementação de um *smart contract*. Normalmente, *oracles* funcionam a partir de uma cascata de execuções que envolvem o disparo de um evento dentro do *smart contract*, seu consumo por

uma entidade exterior, e uma nova chamada à blockchain, com o intuito de disponibilizar os dados solicitados.

Uma preocupação do uso de *oracles* é o que se chama de *The Oracle Problem*, cujo enunciado levanta riscos para o seu uso por se tornarem "pontos únicos de falha", capazes de sofrerem ataques maliciosos ou tornarem-se indisponíveis, prejudicando, assim, a aplicação na blockchain, e também, principalmente, por centralizarem a aplicação em pelo menos um ponto, o que não é aceitável em um ambiente blockchain.

Oracles descentralizados surgem como serviços oferecidos por diferentes projetos como o *Chainlink*, *Paralink* ou o *Witnet*, apenas para citar alguns, que garantem que a fonte de informação a ser consultada seja suficientemente descentralizada para manter sua segurança de uso bem como a adequação aos padrões requeridos por uma aplicação construída em arquitetura blockchain.

3.9 A Ethereum Virtual Machine

Antes de discutirmos a EVM, temos que visitar um conceito da teoria da computação chamado de "Turing Completude". Segundo esse conceito, um sistema é considerado Turing-completo se for capaz de simular uma “máquina universal”. Em outras palavras: um sistema deste tipo deve ser capaz de executar qualquer função programável.

Quando falamos que uma das principais características do Ethereum é o de ser uma rede programável, capaz de executar trechos de código, falamos exatamente sobre a completude de Turing. No caso, o Ethereum é considerado como sendo Turing quase-completo, não por alguma restrição de funcionalidade, mas pelo motivo da existência do *Gas*. Uma conta que não tenha *Gas* disponível não é capaz de executar uma função, sendo esta sua única restrição para ser considerado um “computador universal”.

A EVM é a tecnologia que permite este tipo de abordagem pelo Ethereum. É uma “entidade” virtual, existente em cada um dos nós da rede capaz de interpretar e executar instruções. Tal como ocorre em linguagens de alto nível como o Java, que é compilado pelo Javac para bytecode e executado pela JVM, o Ethereum possui uma linguagem principal, o Solidity, usado para a criação de contratos. Estes são compilados para o bytecode e executados pela EVM.

Por residir em cada um dos nós da rede, a EVM age como um grande supercomputador, cujas capacidades são proporcionais ao número de nós da rede, tal como idealizado por Vitalik Buterin no início do Ethereum.

Tecnicamente, a EVM é montada em uma arquitetura de pilha, com tamanho de palavra de 256 bits. Possui memória volátil em que os itens são endereçados por *arrays*, e provê armazenamento não volátil, na mesma área em que armazena o estado do sistema. Trechos de código são armazenados em uma memória ROM separada, acessada via um conjunto de instruções. A EVM não é, portanto, uma máquina com arquitetura de Von Neumann.

Transações entre contas EOA não são processadas via a EVM. Já quando um contrato é solicitado, ela pode se encarregar de, antes de executar, fazer algumas verificações. Por exemplo, se a quantidade de *Gas* disponível atende à estimativa de uso para execução do código, atesta que os endereços envolvidos sejam válidos, verifica o estado do sistema para garantir que não haverá exceções, entre outras.

Transações são executadas na EVM de forma recursiva, e tanto o estado do sistema como o da máquina são atualizados a cada iteração. Caso alguma exceção ocorra, sua execução é interrompida; caso contrário, a transação é concluída com sucesso.

A EVM possui uma série de instruções comuns, como as que permitem a realização de cálculos aritméticos e controle de fluxo, de uso geral. Como diferencial, possui instruções que dão acesso a

informações sobre endereços de contas, informações de bloco e mesmo sobre preços.

3.10 Ethereum 2.0

O grande sucesso que o Ethereum vivenciou recentemente trouxe consigo uma série de contrapartidas, que foram alvo de pesadas críticas por parte da comunidade e que não foram bem recebidas.

Entre outras coisas, o grande volume de transações sendo incluídas na rede aumentou não apenas os tempos de confirmação mas, também, seus custos associados, além do fato de a crescente dificuldade, necessária para a mineração dos blocos, estar arriscando a entrada da blockchain em uma “era do gelo”, em que seria muito difícil conseguir resolver o quebra-cabeça lógico, impedindo o crescimento da cadeia.

O Ethereum 2.0 é o nome dado a uma série de mudanças e atualizações previstas para ocorrerem na rede, de forma gradual, que visam atuar sobre estas dificuldades, aumentando a escalabilidade da rede como um todo. Apesar de muitas vezes ser referido como um “projeto”, ele não prevê uma nova implementação ou alguma alteração estrutural na rede, como o nome parece sugerir; suas alterações são programadas e graduais e devem ocorrer segundo um cronograma bem planejado.

O tema da escalabilidade é, possivelmente, o mais crucial na versão da Rede Ethereum tal como temos hoje; ele é visto através de um “trilema”, que envolve decisões e contrapartidas que olhem, além da escalabilidade, a segurança e a capacidade de descentralização da rede. São eles:

1. Se, por um lado, o aumento da escalabilidade em um sistema seguro pode ser feito através do aumento das capacidades dos nós individuais, isso traz a contrapartida de diminuir o número

de participantes capazes de atuar como nós na blockchain, diminuindo a capacidade de centralização da rede.

2. Por outra perspectiva, o mesmo sistema escalável, mas descentralizado, aumenta o número de transações incluídas e, como consequência, seu tempo de processamento. Um maior tempo de processamento aumenta a exposição do bloco sendo incluído a ataques sendo, portanto, um problema de segurança que não pode ser ignorado.

3. Já o sistema seguro e descentralizado é exatamente o que temos no Ethereum hoje, que funciona muito bem, porém, que não é escalável e que, portanto, também traz problemas.

A atualização do Ethereum é baseada em uma visão de três componentes que, quando implementados definitivamente, serão capazes de atender a estes três requisitos, sendo traduzidos em um sistema seguro, descentralizado e escalável. São eles: a implementação da cadeia de *Beacon* (*Beacon Chain*), o *merge* da cadeia principal do Ethereum (chamada *mainnet*) com a *Beacon Chain* e o uso de *shards* para processamento dos dados, em 64 novas cadeias, que aumentarão o número de nós participantes e, portanto, aumentarão sua segurança.

Chama-se *Beacon Chain* uma cadeia implementada na rede a partir de dezembro de 2020, que tem como funções principais o armazenamento de dados de registro das validações realizadas na rede, a partir de um consenso de PoS. No futuro, quando as cadeias de *sharding* forem implementadas, elas serão coordenadas pela *Beacon Chain*, que será a espinha dorsal do Ethereum, sobre a qual toda a cadeia será gerenciada, e sobre a qual os blocos serão incluídos.

Apesar de inicialmente a *Beacon Chain* existir separadamente da *mainnet* do Ethereum, em algum momento ocorrerá um *merge* entre as duas, isso é, uma unificação. Uma vez que isso aconteça, pode-se dizer que o Ethereum passe completamente de uma cadeia com

uso do PoW como forma de consenso para uma cadeia com uso do PoS, mais rápido, mais seguro e ecologicamente mais correto, já que não será intensivo em suas demandas energéticas. Um dos desafios do *merge* é o de que ele deverá permitir a execução dos *smart contracts*, algo não previsto no desenvolvimento inicial da cadeia.

O terceiro aspecto do Ethereum 2.0, com previsão de entrada em produção durante o ano de 2023 (com estimativas dadas em fontes não oficiais do projeto para até 2024) inclui a utilização da tecnologia de *sharding*, isto é, a quebra dos dados a serem processados em pequenos “pacotes”, que podem ser distribuídos através da rede e processados independentemente, de forma simultânea.

O planejamento do *sharding* no Ethereum envolve a criação de 64 novas cadeias, através das quais os dados serão distribuídos entre os nós. Ao tratar de *shards*, tanto a quantidade de dados sendo processados como sua distribuição são aumentados de forma bastante acentuada, aumentando a escalabilidade da rede em grande amplitude. Novas cadeias permitirão a entrada de novos nós, o que também aumentará a segurança da rede. Espera-se, ainda, que os *shards* tenham a capacidade de executar *smart contracts*, o que também aumentará o poder da rede como um todo.

Uma vez processados os dados das transações, elas serão incluídas à cadeia principal que, a este momento, será a *Beacon Chain*, finalizando a implementação do Ethereum 2.0.

3.11 O caminho até o Ethereum 2.0

Finalizando nossa apresentação sobre o Ethereum, vamos mostrar uma retrospectiva histórica das principais mudanças ocorridas na rede e introduziremos tópicos sobre o *roadmap* de atualizações

programadas para ela. Sem dúvidas, o momento mais esperado pela comunidade é o lançamento do projeto Ethereum 2.0, que trará muitas mudanças.

Para entender o histórico dos *forks* e das atualizações ocorridas na rede, primeiro temos que conhecer uma figura extremamente importante, surgida um ano após a inauguração do Ethereum: a Organização Descentralizada Autônoma, conhecida pela sigla DAO (*Decentralized Autonomous Organization*).

Uma DAO é uma entidade comumente encontrada dentro do universo blockchain. São organizações que buscam, através de uma forma democrática e aberta de governança, serem autogerenciáveis através de seus membros, e atuam em diversas frentes, tomando decisões sobre projetos e investimentos, por exemplo.

Em 30 de abril de 2016 foi lançada oficialmente uma DAO formada para cuidar de investimentos em Ether. Seguindo os protocolos comuns, ela foi inaugurada através de uma ICO, que vendeu *tokens* de utilização por um período de 28 dias, com um valor arrecadado de cerca de 150 milhões de dólares em menos de um mês de abertura.

Nesse mesmo período, algumas vulnerabilidades foram apontadas no código, que passaram a ser tratadas rapidamente pela comunidade. Um ataque, porém, não pôde ser evitado: nele, um *hacker* solicitou a retirada de seus fundos diversas vezes, de forma acumulativa, sem que o contrato fosse capaz de perceber que os fundos não eram mais existentes. Graças a isso, o ofensor conseguiu solicitar a retirada de cerca de 70 milhões de dólares, conforme cotação do ETH da época.

Esse ataque gerou grandes debates dentro da comunidade. Primeiramente tentou-se remediar o problema através de um *soft fork*, que corrigiria o código, eliminando a vulnerabilidade. Por outro lado, o *hacker* afirmava ter atuado de forma legal, já que o código permitia a transação realizada por ele — e se considerarmos o

código ser um “contrato”, como o próprio nome diz, não haveria aí nada de errado.

Após muita discussão, o *hacker* conseguiu retirar parte de seu dinheiro (cerca de 8 milhões de dólares), e o Ethereum recebeu um *hard fork*, que retomou seu estado para antes da publicação do código da DAO. Assim, em 20 de julho de 2016, no bloco de número 192.000, a cadeia do Ethereum foi dividida em duas: a cadeia original, que hoje é mantida com o nome de *Ethereum Classic* (ETC), um projeto separado, e a nova cadeia, que é utilizada atualmente pelo projeto Ethereum.

Defensores da continuidade do projeto do ETC reforçam que a blockchain deve ser imutável e resistente a censura e que, realmente, o *hacker* apenas se aproveitou de uma opção dada pelo sistema. Ainda que estes argumentos sejam válidos, não há que se negar que, caso o *fork* não tivesse ocorrido, possivelmente o Ethereum não teria evoluído tanto até seu estado atual.

Esse *fork* é conhecido como *The DAO Fork*, e foi, possivelmente, a primeira grande mudança sofrida pelo Ethereum. Ainda no mesmo ano alguns outros *forks* de menor impacto foram realizados, com vistas a aumentar a segurança e a eficiência do sistema.

De acordo com o *roadmap* disponibilizado pela equipe do Ethereum, a partir de 2017 foram implementadas diversas ações em um projeto chamado *Metropolis*, que deveria ser implementado ao longo de vários anos, dividido em três partes, denominadas: *Byzantium*, *Constantinople* e *Istambul* com uma série de melhorias para seu funcionamento, seguido, então, pelo projeto *Serenity*, com vistas à adoção do Ethereum 2.0.

Em 16 de outubro de 2017 o *Byzantium* foi implementado. Nele, entre outras coisas, começaram-se a implementar algumas mudanças estruturais para a melhoria da escalabilidade da rede a partir do trabalho na camada 2, já discutido anteriormente, além de incluir atributos de segurança e privacidade. Ele também diminuiu o

valor pago como recompensa pelo processo de mineração, que então era de 5 ETH e que passou para 3 ETH por bloco.

O *Byzantium* foi fundamental para permitir um ambiente mais propício para o uso de *smart contracts* de forma mais ampla, o que trouxe não apenas benefícios para desenvolvedores, como também foi muito bem recebido por investidores.

A atualização *Constantinople* foi implementada em 28 de fevereiro de 2019. Segundo planejamento do projeto Ethereum, ela deveria incluir alterações de melhoria de desempenho na rede, porém, mais importante do que isso, deveria começar a facilitar o processo de mineração, que desde essa época vinha se tornando muito difícil — um fenômeno conhecido como "bomba de dificuldade" (*difficulty bomb*), uma ameaça à viabilidade do funcionamento do Ethereum.

Ao contrário do esperado, o *Constantinople* não foi capaz de resolver este problema, apesar de ter atuado em melhorias importantes da rede, preparando o terreno para a adoção do PoS.

A atualização *Istambul* foi finalmente disponibilizada em 08 de dezembro de 2019, trazendo cerca de trinta pacotes de melhoria para a rede, envolvendo aspectos de performance, otimização de custos computacionais e a possibilidade de uso do *Zcash* com o Ethereum, por exemplo.

Ele também não foi capaz de atuar no problema da bomba de dificuldade da rede; isso ocorreu apenas em 02 de janeiro de 2020, com a subida das melhorias contidas no pacote *Muir Glacier*, que mais uma vez retardou o efeito da dificuldade da mineração, ainda que não o tenha resolvido.

Uma das principais inovações ocorridas com vistas à adoção do Ethereum 2.0, as cadeias de *Beacon* foram implementadas com atualização ocorrida em 1 de dezembro de 2020, quando foi ao ar seu bloco gênese em uma atualização chamada *Beacon Chain Genesis*.

O projeto *Serenity*, atualmente em curso e sucessor do *Metropolis*, é o que pretende implementar o Ethereum 2.0. Ele é composto por diferentes fases, e envolve uma série de atualizações programadas, denominadas *Berlin*, *London* e *Shanghai*.

O *upgrade Berlin* entrou em produção em 15 de abril de 2021. Dentre suas principais funcionalidades destacaram-se a diminuição dos custos da taxa de *Gas*, isto, em preparação para futuras atualizações.

Em 05 de agosto do mesmo ano, a atualização *London* foi publicada. Entre as principais funcionalidades incorporadas estiveram a implementação do EIP-1559, que previa a atualização do mercado de taxas e pagamentos em *Gas*, a implementação do EIP-3529, que reduziu as taxas de *Gas* para operações na EVM, e a EIP-3554, que deu uma sobrevida para a utilização do PoW até, pelo menos, dezembro de 2021, evitando a entrada na chamada "era do gelo", devido à bomba de dificuldade, apresentada anteriormente.

Esse prazo foi estendido graças à atualização *Arrow Glacier* (09 de dezembro de 2021) que, mais uma vez, se preocupou com este aspecto, dando uma sobrevida ao Ethereum.

Até a migração para o Ethereum 2.0, outros *upgrades* serão necessários para trabalhar nesta frente, que não trarão novas funcionalidades para a cadeia, mas que permitirão a continuação de sua existência.

> Uma EIP é uma proposta de melhoria do Ethereum, abreviação de *Ethereum Improvement Proposal*.

Pouco mais de 2 meses após o *London*, em 27 de outubro de 2021 a atualização *Altair* entrou produção, sendo o primeiro *upgrade* programado com atuação na *Beacon Chain* e, provavelmente, o

último pacote de atualizações programadas disponibilizado antes da migração para o Ethereum 2.0.

Com o *Altair* foram implementadas melhorias baseadas na EIP-2982, que incluíram o suporte aos chamados "clientes leves", isto é, "nós com poucos recursos computacionais", e políticas de penalizações a usuários que não sigam as boas práticas de participação recomendadas pelo Ethereum.

> A atualização *Shanghai*, quando realizada, completará a passagem do mecanismo de consenso do PoW para o PoS, concluindo a adoção do Ethereum 2.0!

Conclusão

Neste capítulo conhecemos com detalhes o funcionamento do Ethereum. Vimos que, ainda que muito de sua estrutura seja similar ao modelo de blockchain mais popularmente utilizado como base — do projeto Bitcoin —, temos diferenças importantes em questões de construção de blocos e temas de criptografia, por exemplo, além da própria existência da EVM — que por si já justificaria a existência deste capítulo, como um diferencial importante.

A adoção do Ethereum 2.0 (visto nas partes finais do capítulo) é um processo de mudança lento e gradual, porém, com fases bem definidas, de acordo com um cronograma constantemente atualizado pelo time do projeto, e que trará grandes benefícios para a Rede. Ele trará soluções para temas críticos — como a questão da escalabilidade — que são gargalos para o consumo da blockchain em algumas situações, e que, quando solucionados, permitirão ao projeto alcançar novos níveis de operação.

Não é demais notar que, mesmo nesse cenário, o funcionamento do Ethereum seguirá tal como ocorre hoje (logicamente, com algumas mudanças esperadas, como a questão da adoção do *PoS* como solução de consenso) e que, portanto, não se pode falar aqui da

criação de um novo projeto, mas apenas da existência de atualizações pontuais no protocolo.

Agora que entendemos como a estrutura do Ethereum funciona, podemos começar a pensar em como utilizá-lo para a solução de problemas reais. Estes serão abordados no próximo capítulo, em que trataremos sobre os tipos de aplicações que utilizam o Ethereum como base, e os tipos de cenários em que são aplicados.

Capítulo 4
Anatomia das aplicações Ethereum

4.1 Introdução

A partir de agora passaremos a estudar a anatomia das aplicações que utilizam a estrutura blockchain para existir. Até então entendemos como a Rede Ethereum funciona, vimos seus principais fundamentos, modos de trabalho e os papéis envolvidos.

Deste ponto em diante, entenderemos como a cadeia é utilizada como estrutura para a sustentação de aplicações, o que exige um cuidado especial com várias de suas particularidades. Apresentaremos temas pertinentes sobre aplicações e integração de blockchains, que utilizam ferramentas e conceitos que possivelmente ainda não são familiares a devs que não tenham experiência com esse tipo de aplicação.

De forma geral, chamam-se “Aplicações Descentralizadas” (dApps, *decentralized Applications*) quaisquer aplicações que utilizem um *smart contract* como back-end, podendo o front-end estar, ou não, presente e ser, ou não, distribuído.

Devido às suas características arquiteturais, uma dApp possui necessidades importantes e casos de uso muito específicos. Ao passo que uma dApp se aproveita das vantagens oferecidas pela blockchain no que tange à privacidade de integridade de dados, por exemplo, ela também é prejudicada por fatores como excessivos tempos de latência. Lembremo-nos que o Ethereum é capaz de processar cerca de um bloco a cada quinze segundos, o que pode se tornar um gargalo em determinados cenários.

Além disso, *smart contracts* são imutáveis, o que exige uma série de cuidados no momento de sua criação, já que realizar sua manutenção é um desafio importante. Logo, testes unitários tornam-

se tão ou mais importantes que verificações de segurança e a adoção de protocolos de boas práticas.

Algo curioso no trabalho com dApps é que elas possuem um sistema especial de gestão de nomes de acesso, similar ao DNS (utilizado em páginas e aplicações centralizadas), chamado de *Ethereum Name Service* (ENS). Isso também deve ser levado em conta, visto haver uma etapa a mais para seu desenvolvimento.

Uma categoria de dApp que vem ganhando grande notoriedade nos últimos tempos é a chamada de *Finança Descentralizada* (DeFi) que, como vista anteriormente, é desenvolvida com o intuito de possibilitar a oferta de serviços financeiros sem a presença da figura centralizadora. A ideia por trás da DeFi é a de utilizar as propriedades de segurança e imutabilidade da blockchain para armazenar, gerenciar e oferecer soluções que envolvam transações financeiras de forma facilitada.

Tokens do tipo ERC-721, isto é, NFTs, são um terceiro tipo de aplicação na Rede Ethereum que tem ganhado atenção. Conforme visto no capítulo anterior, um NFT se diferencia de um *token* fungível por não dar direito ao usufruto de um serviço, ou acessos exclusivos, por exemplo. Ao contrário, seu valor advém simplesmente de sua posse.

Em grosso modo, NFTs funcionam como “títulos de propriedade”. Por serem únicos e imutáveis eles garantem ao possuidor o direito de propriedade de ativos digitais, o que traz implicações para o mundo jurídico e, portanto, são também transacionados, criando um nicho de mercado bastante expressivo. No início de 2021, por exemplo, a casa de leilões Christie’s foi manchete na imprensa mundial após transacionar, pela primeira vez, um NFT, arrecadando cerca de 70 milhões de dólares ao arrematar uma obra de arte do artista digital Beeple — que passou a se referir à sua obra como uma “criptoarte”.

O processo de criação de um NFT envolve, além do objeto digital, de uma etapa de “tokenização”, feita no Ethereum, e que será vista mais à frente neste capítulo.

Finalizando, veremos como estão sendo desenvolvidas as soluções que tentam resolver o problema da escalabilidade no Ethereum, que incluem o uso de outras redes de forma paralela à *mainnet* do Ethereum. Um dos projetos comerciais desenvolvidos para isto é o Polygon, um protocolo — e também autodenominado *framework* — que permite a integração de cadeias Ethereum, e que também permite criar um subsistema dentro da Rede — apelidado de *Internet das Blockchains*.

4.2 Smart Contracts

O nome “*smart contract*” é dado a qualquer programa ou trecho de código que seja desenvolvido e rode na Rede Ethereum. No geral, um contrato é composto por uma coleção de ações (chamadas funções) e dados, que alteram o estado da blockchain e que são responsáveis por fazê-la chegar ao estado atual.

Como visto, um *smart contract* é ele mesmo um tipo de conta no Ethereum e é capaz de criar transações ao longo da rede, com o revés de não ser capaz de iniciar uma transação por conta própria. Por residir na blockchain, ele possui um endereço conhecido e registros de transações associadas — tendo, portanto, um fluxo de caixa financeiro associado.

A ideia da existência de um contrato inteligente é mesmo anterior ao advento da blockchain, datando da década de 1990, quando o termo foi cunhado por Nick Szabo — que o definiu como “um conjunto de garantias especificadas em forma digital”. Szabo fez sua descrição em publicação intitulada *Formalizing and Securing Relationships on Public Networks* (“Formalizando e Assegurando Relacionamentos

em Redes Públicas”, em tradução livre), em que apresentou a ideia do “autômata como autoridade".

Nela, o autor discorreu sobre as dimensões do desenho de um contrato, já prevendo a ideia dos custos de transações, padrões de observabilidade, verificabilidade e proteção de terceiros, e também sugeriu o uso de criptografia e soluções de autenticação como formas de segurança.

Nessa mesma publicação ele citou a máquina de vendas como um *smart contract* primitivo: qualquer pessoa que possua moedas pode usá-lo, e suas cláusulas são imutáveis — troca-se o valor inserido em moedas por um produto à venda (caso o valor inserido seja igual ou maior ao valor de venda), e devolve-se o troco, caso seja necessário. Não existe outra possibilidade de interação, exatamente como em um verdadeiro contrato.

Um *smart contract*, no entanto, segundo Szabo, vai além: ele age sobre propriedades de forma dinâmica e permite maior poder de observação e verificação do que o que ocorreria em uma máquina de refrigerantes, por exemplo.

Tecnicamente, a criação de um contrato inteligente é uma transação na Rede Ethereum, da mesma forma que suas invocações e que, portanto, também necessitam de *Gas* para funcionar. Devemos nos lembrar, ainda, que transações que utilizam *smart contracts* necessitam da máquina virtual do Ethereum. Em poucas palavras, um contrato é, portanto, um programa compilado para o *bytecode*, interpretado pela EVM e, como tal, possui linguagens de programação e compiladores específicos.

As duas principais linguagens de alto nível utilizadas para o desenvolvimento de *smart contracts* são o Solidity, visto em detalhes em capítulos posteriores, orientada a objetos e influenciada pelo C++, e o Vyper, esta baseada no Python. Programadores mais experientes podem desejar interagir com os contratos com linguagens de mais baixo nível, principalmente para cenários que

envolvam segurança e otimização de código; nesses casos, temos linguagens como o Yul e o Yul+ que podem ser usadas livremente, e que dão acesso, inclusive, ao *WebAssembly* do Ethereum — chamado *Ewasm*.

Uma vez compilado para o *bytecode* da EVM, o *smart contract* também gera uma estrutura, chamada ABI (*Application Binary Interface* - Interface do Binário da Aplicação) —, composta por um JSON que expõe suas propriedades e que o permite ser utilizado por clientes, como o JavaScript. Maiores detalhes sobre o uso da ABI serão dados mais à frente neste livro.

Uma importante característica, a composibilidade é regra-geral no trabalho com contratos. Por residirem na Rede, *smart contracts* podem ser acessados externamente por diferentes meios e aplicações, inclusive por outro *smart contract*, mesmo que este tenha sido desenvolvido por terceiros, sem qualquer tipo de relação com o autor.

Ainda que existam técnicas aplicadas ao código que podem impedir a origem de acessos a certas funções, há que se levar em conta que este é um importante padrão de design de sistemas, e que traz grandes benefícios quando utilizado corretamente. Já uma limitação é sua impossibilidade de realizar chamadas HTTP diretamente, o que pode ser contornado de diversas maneiras (inclusive por meio dos *oracles*).

As propriedades de um contrato, chamadas de “dados”, podem ser declaradas como pertencendo a dois locais: na memória, em que são perdidas logo após o uso, ou no "armazenamento" (*storage*), em que são armazenadas permanentemente na blockchain — e, devido a sua imutabilidade, ali residem para sempre. Maiores detalhes serão dados em capítulos posteriores, porém, para o momento, basta saber que isso deve ser levado em conta quando se desenha um contrato, visto que os custos associados ao tipo de armazenamento variam muito, sendo a utilização em memória muito mais barata.

Funções, por sua vez, representam as ações tomadas pelo contrato. Estas podem ser internas ou externas, públicas ou privadas, e se diferenciam pela sua capacidade de serem invocadas dentro ou fora de seus contratos de origem ou seus derivados. Podem ainda ser puras, pagáveis (*payable*) ou apenas de consulta, de acordo com seu comportamento esperado — funções pagáveis, por exemplo, são capazes de debitar valores da conta de quem a está acessando simplesmente pelo uso de uma palavra reservada em sua declaração.

O desenho de um *smart contract* deve ser muito bem elaborado, visto que todas as suas etapas, da publicação à utilização, envolvem custos — pagos tanto na forma de *Gas* como em cobranças feitas diretamente pelas funções. Somando-se a isso, fatores como a imutabilidade do código e sua publicidade podem causar experiências desagradáveis caso não sejam bem planejados. Lembremo-nos, por exemplo, do ataque ao DAO que causou a cisão do Ethereum em dois projetos separados e um prejuízo de alguns milhões de dólares devido a um *bug* em um contrato que havia sido publicado.

Boas práticas no desenvolvimento de *Smart Contracts*

Se por um lado as características dos *smart contracts* podem ser intimidadoras, considerando, principalmente, as questões financeiras e de segurança envolvidas, por outro, a comunidade as entende como um incentivo à adoção de boas práticas e otimização de código, ambas visando à diminuição dos riscos associados.

Tendo isso em vista, diferentes frentes atuaram de modo a compilar guias para auxiliar no desenvolvimento; vários deles são citados nas referências do capítulo final deste livro.

Um exemplo de boas práticas são os chamados *Clean Contracts* (Contratos Limpos), que fazem alusão ao modelo de trabalho do *Clean Code*, muito conhecido por desenvolvedores, e cujo modelo

foi aplicado em nossos capítulos dedicados ao Solidity e à escrita de contratos. Entre outras coisas, ele define especificações sobre:

1. Nomenclatura de contratos, funções e variáveis.
2. Ordenações de escrita, declarações dentro de funções, declarações de funções e modificadores.
3. Modos de documentação e formatação do código.
4. Uso de ferramentas de análise estática.
5. Uso de estruturas de dados.
6. Manipulação de exceções.
7. Definição de testes unitários e de integração e cobertura de testes.
8. Análises de segurança e ferramentas de análise de código.
9. Otimização do uso de *Gas*.
10. Integração contínua.

Existem ainda boas práticas não oficiais, que lidam com os cuidados a serem tomados ao utilizarmos chamadas externas, como utilização de métodos com menor vulnerabilidade ou custos associados, mudanças de estado do sistema, uso de verificações intermediárias, criação de modificadores personalizados, uso de interfaces e contratos abstratos, assim como o monitoramento, trabalho com eventos e geração de logs.

Dentre as práticas de engenharia de software, talvez a que mais faça sentido no universo dos *smart contracts* seja o “prepare-se para a falha” (*prepare for failure*). Isso porque os contratos não podem ser modificados, e o custo de uma falha na execução pode ser inestimável.

Se, por um lado, a aplicação de padrões como o *circuit breaker*, capaz de proteger o contrato contra os efeitos de uma falha, é uma prática de desenvolvimento, a tomada de decisões em momento de codificação também possui efeito extremamente positivo para o desenho do contrato.

A limitação do número de transações realizadas em um intervalo de tempo, ou a ordem de declarações dentro do contrato, exemplos de tomadas de decisão em momento de codificação, podem trazer ganhos de segurança importantes.

Testando *Smart Contracts*

Por fim, não podemos terminar de falar sobre desenvolvimento de *smart contracts* sem citar a existência de blockchains usadas para o teste — as chamadas “*testnets*” (do inglês, *test network* — "rede de teste", em tradução livre). Essas redes são apartadas totalmente da rede principal (a *mainnet*, vista anteriormente) e possibilitam simular o funcionamento de um contrato com grande grau de realismo, desde sua publicação até seu uso.

Durante o trabalho com uma *testnet*, trabalhamos com moedas e valores fictícios, que permitem realizar validações sobre tempos e custos, sem ônus para o desenvolvedor. Há de se tomar cuidado aqui para não realizar transferências reais de Ether por engano, o que seria um desperdício de dinheiro.

Smart contracts são a parte mais importante do desenvolvimento de aplicações em blockchain. Isso porque são o centro de todo o programa, responsáveis pela camada lógica propriamente dita, e por serem eles o elemento que confere a característica da descentralização. Devido à sua posição crítica em uma arquitetura blockchain, devem ser planejados e desenvolvidos com extremo rigor, além de monitorados de perto para quaisquer desvios não previstos.

Apesar de termos citado mecanismos e práticas a serem levadas em conta durante a fase de seu desenvolvimento, fizemos uma abordagem muito genérica, apenas citando os principais fatores a serem considerados. Em um livro que se propõe a apresentar fundamentos de desenvolvimento, como o nosso, não seria apropriado e nem possível entrar neste assunto com uma riqueza

muito grande de detalhes. Por isso, deixamos algumas citações nas referências que poderão servir como direções para estudos mais avançados neste tema.

Mais para a frente, quando introduzirmos a linguagem de programação Solidity, revisitaremos alguns dos temas aqui citados, o que dará uma clareza ainda maior no entendimento do funcionamento deste tipo de artefato.

4.3 Aplicações descentralizadas (dApps)

Se na seção anterior descrevemos o funcionamento dos *smart contracts*, a partir de agora nos focaremos nas aplicações que os utilizam como tecnologia principal, porém, que também são capazes de descentralizar outros aspectos, tais como o armazenamento ou a camada de troca de mensagens.

O conceito de dApp é bastante abrangente. De acordo com a documentação oficial do Ethereum, uma dApp é qualquer aplicação que combina a utilização de um *smart contract*, descentralizado, rodando em uma rede P2P, e um *front-end*, o que abre brechas para uma infinidade de tipos de aplicações, categorizadas pela comunidade como dApps verdadeiras ou não.

De forma bastante ilustrativa, diz-se que o papel de um *smart contract* em uma dApp é o mesmo daquele desempenhado pelo back-end em uma arquitetura de microsserviços, em que o front-end é desacoplado e pode ser construído, virtualmente, com qualquer tecnologia. Neste ínterim, o papel do banco de dados seria representado pelo próprio *ledger* do blockchain, o que completa nossa comparação.

Tal como um *smart contract*, uma dApp também se aproveita das vantagens oferecidas pela tecnologia blockchain no que diz respeito a privacidade, integridade e segurança, além de se tornar totalmente

disponível — exceto, talvez, caso haja algum tipo de indisponibilidade com a camada de apresentação. Da mesma forma, algumas desvantagens surgem de seu uso, também inerentes à blockchain.

Por exemplo, sua manutenção é dificultada, uma vez que o código do contrato não pode ser alterado. Fatores como grandes tempos de latência e baixa escalabilidade também devem ser lembrados por, possivelmente, impactarem no sucesso da aplicação. Além disso, a necessidade do uso de *tokens* para seu consumo pode gerar dificuldades iniciais para o usuário desacostumado com este tipo de tecnologia.

Apesar de não ser a única blockchain que permite a publicação de dApps, o Ethereum é a principal delas, tendo representado, em 2020, ser a blockchain escolhida para ser utilizada por 82% das dApps, com um volume financeiro estimado em 12 bilhões de dólares somente no primeiro semestre daquele ano, segundo artigo publicado pelo Decrypt (disponibilizado nas referências). Em número de usuários diários, no entanto, o Ethereum oscila entre as primeiras posições.

Uma rápida pesquisa na internet é capaz de mostrar a diversidade de aplicações construídas como dApps, indo de aplicações de jogos eletrônicos, apostas, e aplicações financeiras, até usos em manutenção de identidade, aplicações de saúde, governança e mídia.

Clientes Ethereum

Para que uma aplicação consiga interagir com a Rede Ethereum ela deve, primeiro, ser capaz de se conectar com um nó pertencente à blockchain. Esta interação é facilitada através de uma especificação padronizada de uma interface dos clientes Ethereum denominada JSON-RPC — um protocolo agnóstico que pode ser transportado através de diferentes mecanismos de comunicação, além do HTTP. A conexão com o nó, no entanto, é um pouco mais complexa.

Existem diferentes formas de se conectar diretamente com um nó na Rede Ethereum. Isao pode ser conseguido através de clientes amplamente conhecidos como o Geth (*Go-Ethereum*) ou o Parity, que funcionam muito bem, porém, que dotam a pessoa desenvolvedora de ampla gama de responsabilidades no que diz respeito à manutenção dos dados da Rede, incluindo sincronismos.

Aqui surge uma particularidade importante: quando acessamos um cliente Ethereum, não apenas baixamos toda a blockchain diretamente em nossa máquina, do bloco gênese até o último bloco adicionado (o que por si representa uma quantidade considerável de dados, cujo tamanho deve ser dimensionado, ao optar por uma infraestrutura adequada para isso). Quando instalamos um cliente do Ethereum e fazemos seu sincronismo completo, também precisamos validar cada um dos blocos, desde o início, o que é uma tarefa extremamente demorada.

De fato, uma vez que no blockchain os dados são imutáveis, a cadeia contém todos os registros históricos ocorridos nela registrados como transações. No início do Ethereum uma série de tentativas de ataques de DoS foi tentada contra a Rede. Esse fato é percebido facilmente ao sincronizarmos a rede inteira pela primeira vez, quando passarmos pela faixa de blocos do 2.283.397 até o 2.700.031 e percebemos, claramente, uma queda no ritmo de validação, que se torna extremamente lenta, como resultado dos dados registrados para esses ataques.

Em alguns casos, podemos preferir acessar os chamados “clientes remotos” (*remote clients*), que são uma versão de acesso aos nós que provêm menos funcionalidades, porém, que acontece de forma mais rápida; isso é útil, por exemplo, quando falamos de acesso via dispositivos com menor capacidade, como o uso de clientes em carteiras mobile ou ambientes de desenvolvimento e teste.

Além de uma série de requisitos e procedimentos técnicos, a decisão de participar do Ethereum como um nó passa pela necessidade de hardware bastante robusto. Dependendo do cliente

e do tipo de sincronismo selecionados, só os requisitos de memória chegam a ser da ordem de 7 TB, com SSD de 2 TB disponíveis e 32 GB de RAM. Estes fatores devem ser levados em conta quando escolhemos um *client*, além dos modos próprios de sincronismo e as redes às quais ele é capaz de acessar.

Algumas soluções comerciais disponibilizam este acesso via API, o chamado *Node as a Service*, em que uma taxa é cobrada pelo consumo dos endpoints disponibilizados. Uma solução bastante conhecida deste tipo, que abstrai toda a complexidade inicial de acesso, sincronismo e validação é o Infura, que oferece uma suíte de soluções através de sua Ethereum API e permite a realização da comunicação com a blockchain de forma facilitada, inclusive para soluções com acesso a camadas apartadas com vista aos projetos de escalabilidade no Ethereum (vistos mais à frente neste capítulo). Outros fornecedores importantes aqui são o Alchemy e o QuickNode, apenas para citar alguns.

Tratando de código, especificamente, a documentação do Ethereum cita um grande número de bibliotecas disponíveis para a interação com a Rede. Elas incluem funcionalidades que permitem desde a conexão com a blockchain até a realização de consultas específicas sobre a estrutura, tais como números de blocos, saldos disponíveis, conversões entre unidades do Ether e consultas endereços, bem como a realização de consultas mais complexas utilizando GraphQL, por exemplo.

Interação com aplicações e acesso

Da mesma maneira, linguagens *back-end* de uso geral possuem bibliotecas próprias. Desenvolvedores .NET podem se beneficiar do Nethereum para integrar soluções blockchain às suas aplicações, enquanto linguagens JVM como o Java ou o Scala têm a opção de utilizar o web3j, apenas para citar algumas opções.

Provavelmente a biblioteca mais amplamente utilizada seja a web3.js, uma API JavaScript para a interação com um nó Ethereum,

extremamente madura. Ela possui uma coleção de módulos que contêm soluções para diversas funcionalidades para o universo do Ethereum, tais como a interação com *smart contracts*, protocolos para a realização de *broadcasting* na rede P2P, acesso a sistemas descentralizados de armazenamento de arquivos, entre outros, prontos para interagir com o JSON-RPC dos clientes da Rede.

Suas chamadas tornam-se tão simples que chegam a ser comparadas com a realização de requisições AJAX em um front-end para a consulta de informações; neste caso, consultando diretamente a blockchain.

A comunicação com o contrato, especificamente, é realizada através da ABI, apresentada anteriormente neste capítulo. Uma ABI é uma interface que representa as propriedades de um contrato, dando informações como tipos de dados esperados, funções existentes, retornos, entre outros. Ao ser incorporada no código (por vezes passada como parâmetro em funções no web3.js), ela permite a chamada a funções diretamente nos *smart contracts*, tal como um serviço externo que é incorporado.

O front-end da dApp, como visto, pode ou não ser descentralizado. Isso pode ser conseguido, por exemplo, através do uso de mecanismos de armazenamento distribuídos do Ethereum, que serão vistos com detalhes ao final deste livro. Para o momento, basta saber que isso é uma possibilidade.

Um dos principais desafios da especificação de uma dApp são as definições sobre a governança de dados, um ponto crucial.

Se, por um lado, a falta de uma figura de autoridade centralizadora, com poder de controle nas transações, representa o ideal da blockchain, sua ausência também pode expor a fragilidade de uma aplicação. O próprio ataque ao DAO foi um exemplo das consequências que podem aparecer quando ninguém é capaz de arbitrar os fatos ocorridos dentro da aplicação, e isso deve ser muito bem ponderado em tempo de desenvolvimento.

Finalizando, devemos dedicar algumas linhas para apresentar o conceito do *Ethereum Name Service* (ENS), que implementa um controle de nomes na rede, facilitando o acesso, também de forma descentralizada. A ideia do ENS surgiu na própria publicação do Ethereum quando, já em seu *whitepaper*, apresentava-se uma solução de registro de nomes. Ele mesmo é uma dApp de livre acesso que controla o registro de nomes, fazendo a conversão de seus endereços lógicos (apresentados na forma de números hexadecimais), para nomes de fácil leitura humana — exatamente como o DNS opera com endereços IP.

Por ser um sistema descentralizado ele exige um mecanismo de consenso para determinar se o nome solicitado está realmente disponível, uma vez que esta escolha deve ser propagada e validada por toda a rede. Isso é feito através de um mecanismo de leilão, utilizando o padrão *Vickery*: os participantes apresentam seus valores de forma secreta e, no momento da abertura das propostas, vence quem tiver dado o maior lance — pagando, porém, o valor da segunda maior proposta apresentada.

Cada uma das etapas do leilão, entre a disponibilidade do nome, apresentação de propostas e a finalização, ocorre em um intervalo de tempo relativamente longo, o que garante que todas as transações na blockchain já tenham sido confirmadas até o final do processo, eliminando eventuais conflitos entre diferentes nós.

4.4 Finanças descentralizadas (DeFi)

O termo DeFi é definido na documentação oficial do Ethereum como sendo “um sistema (financeiro) aberto e global construído para a Era da Internet”, apresentado como uma alternativa ao sistema atual, “burocrático e engessado”. Ele é um termo coletivo para a oferta de serviços e produtos financeiros, acessíveis a qualquer pessoa que utilize a blockchain.

O DeFi surge como uma consequência direta e orgânica do florescimento da oferta de serviços econômicos. Se durante o século passado a automação de processos (através do uso de computadores) e a revolução causada pelo uso da internet foram os principais motivadores da indústria, a partir do século XXI esse movimento ganha novos rumos com o surgimento das *Fintechs*, que facilitaram a oferta de muitos dos serviços de forma eficiente. Elas são capazes de concorrer diretamente com grandes corporações, tradicionais, concentradoras de capital — características, estas, que até hoje inibem o surgimento de novos atores no mercado e diminuem a capacidade de inovação neste nicho.

Ainda que a adoção de tecnologia tenha melhorado (e muito), a experiência de oferta de serviços para o consumidor, o sistema financeiro como um todo, no entanto, ainda é percebido como ineficiente. Somam-se a isso o fato de decisões macroeconômicas afetarem diretamente um grande número de pessoas — para o bem ou para o mal — e o fato de elas, na maior parte das vezes, estarem ligadas a fatores políticos.

A aparição de uma economia baseada em criptoativos durante a última década democratizou, em partes, o mercado financeiro. Grosso modo, qualquer pessoa com acesso à internet, hoje, é capaz de movimentar criptomoedas e ativos digitais de forma livre, conforme sua vontade e sem restrições como aquelas impostas por demandas de crédito, históricos financeiros, ou mesmo imposições políticas — lembremo-nos, por exemplo, do confisco das cadernetas de poupança ocorrido no Brasil na década de 1990.

A criptoeconomia surge como um movimento que permite não apenas a quebra de barreiras físicas e políticas, como também econômicas e mesmo jurídicas, considerando, por exemplo, a dificuldade surgida por regulamentações divergentes sobre envio e recebimento de divisas. Tudo isto, com base nas características oferecidas pelo blockchain como tecnologia segura e com alto grau de privacidade.

Neste cenário, o DeFi surge como um movimento natural. Ao ignorar questões como identidade, origem, localização e histórico e, estando baseada simplesmente em regras predefinidas na forma de *smart contracts*, de conhecimento geral, aplicações financeiras descentralizadas permitem o acesso igualitário à oferta de produtos e serviços financeiros disponíveis na rede.

Uma descrição bastante interessante do DeFi é a de ser um “dinheiro programável”, já que através da criação de *smart contracts* torna-se possível realizar a movimentação de criptomoedas e ativos digitais de forma automática, bem como a realização de negociações e acompanhamento do mercado.

Um DeFi é, basicamente, uma dApp criada para oferecer um serviço financeiro (e, portanto, também trabalha com *smart contracts* em sua camada lógica). Logo, a ele se aplicam todas as características apresentadas anteriormente. Ele é hoje o mercado com maior crescimento em blockchain.

Uma medida de valor muito utilizada neste âmbito é o TVL (*Total Value Locked*, "Valor Total Bloqueado"), que contabiliza o valor total depositado em *smart contracts* por meio de *tokens*. Somente no primeiro trimestre de 2021 estima-se que ele tenha subido cerca de 5 vezes, para aplicações DeFi. Dados mais precisos citam um crescimento de um TVL total de 1 bilhão de dólares em abril de 2020 para cerca de 32 bilhões de dólares em fevereiro de 2021, um crescimento de mais de 32 vezes em menos de um ano, segundo apresentação feita pelo Finematics (link disponível nas referências).

Dados mais recentes estimam o valor do TVL em projetos DeFi para outubro de 2021 na ordem de 218 bilhões de dólares — um crescimento de 1000% em 2021 —, estando o Ethereum com 69% deste mercado, segundo o Crypto News Australia.

Quando analisamos o volume movimentado por casas de câmbio descentralizadas (DEX — *Decentralized Exchange*), comparamos um crescimento de movimentação de meio bilhão de dólares em

abril de 2020 para 50 bilhões de dólares em janeiro de 2021, o que por si dá uma boa ideia de como o mercado de DeFi vem ganhando cada vez mais importância.

Ao termos citado a existência das DEX no parágrafo anterior, começamos a ter uma ideia da dimensão do DeFi: de fato, o termo DEX é um pouco controverso por se relacionar, em teoria, a corretoras de valores, e não à operação de câmbio, propriamente dito. Parte das críticas sobre esta nomenclatura diz respeito à impossibilidade de existir uma “organização corretora de valores” dentro do mundo descentralizado, uma vez que sua mera existência já indica uma centralização. A ação de câmbio, no entanto, esta sim pode ocorrer de forma descentralizada, através de *smart contracts* e, por isso, nossa opção por traduzi-la assim.

O ecossistema do DeFi é um universo, impossível de ser estudado e detalhado por estar em constante expansão. Adiciona-se aqui uma camada de complicação, que é o fato de que DeFis existem em diferentes blockchains e não só no Ethereum e que nelas podem existir diferenças importantes.

É possível, ainda, que uma mesma oferta de DeFi exista em mais de uma blockchain de forma simultânea, ou que seja migrada.

Recentemente o Ethereum viu algumas de suas DeFis migrarem para blockchains concorrentes devido às altas taxas de transação em *Gas* e a demora nas transações. É o caso do SushiSwap, uma das bolsas de investimento descentralizadas de maior sucesso do Ethereum, que indicou ter iniciado um processo de migração para a plataforma Avalanche, blockchain dedicada à hospedagem de aplicações financeiras.

De modo abrangente, não existem limitações para a oferta de soluções DeFi, dependendo estas apenas da imaginação de seus criadores. A seguir veremos algumas das principais categorias sendo ofertadas na Rede Ethereum e comentaremos sobre os serviços mais populares de cada um.

Importante esclarecer que a mera citação destes não é feita como forma de propaganda nem incentivo ao uso, muito menos como recomendação de investimento, sendo feita exclusivamente com fins didáticos por acreditarmos que aplicações maduras e populares, com aceitação da comunidade, sirvam como exemplo de como o assunto está sendo levado em projetos reais.

Projetos em DeFi

Um dos primeiros termos com o qual nos deparamos quando começamos a ganhar familiaridade com o DeFi é o chamado *stablecoin* (“moeda estável”). Uma *stablecoin* é uma criptomoeda tal como o Ether ou o Bitcoin, porém, ao contrário destas, possui uma volatilidade menor no mercado e, portanto, confere maior poder de segurança no que diz respeito ao valor.

Stablecoins podem ter seus valores atrelados a diferentes referências, tais como moedas “reais” (como o Dólar, o Euro ou o Real), a *commodities*, como o ouro ou o petróleo ou ainda em outras criptomoedas. Elas são úteis em cenários em que se deseje ter uma ideia de paridade de valor, em que o intuito não é o investimento em si, mas seu usufruto. Por exemplo, seu uso fora do sistema financeiro tradicional ou como forma de proteção, em países com altos níveis de inflação.

Uma das *stablecoins* mais conhecidas e controversas, o Tether (USDT) tem como premissa a paridade com o Dólar Americano, tendo seu preço praticamente estável em 1:1. Para funcionar em uma economia, espera-se, a existência de um caixa para garantir essa paridade, o que gera muita desconfiança nesse projeto, porém não diminui seu sucesso. O Real Brasileiro, por sua vez, possui uma *stablecoin* construída na Rede Ethereum: o CryptoBRL (cBRL), desenvolvido utilizando o padrão de *token* ERC-20.

Uma DAO da Rede Ethereum chamada MakerDAO que se propõe a oferecer soluções de crédito e empréstimos apresenta um caso de uso importante de uma *stablecoin*: para contornar os problemas que a volatilidade do ETH causaria na segurança de suas operações, ela passou a trabalhar com sua própria *stablecoin*, o DAI, com paridade com o Dólar Americano.

Citando empréstimos em DeFI, o Compound é outro projeto criado em cima do Ethereum, que permite tanto a oferta como a tomada de empréstimos em criptomoedas. Uma pessoa que possua algum valor em criptomoedas que deseje disponibilizar para empréstimo, recebendo um valor de juros em retorno, utiliza seus *tokens* COMP, construídos com base no ERC-20. A plataforma se encarrega de conectar credores a tomadores e ajustar, de forma dinâmica, as taxas aplicadas, com base em uma relação de oferta e procura.

Um importante projeto DeFi Ethereum, o Uniswap, atua dentro dos chamados AMM (*Automated Market Makers*, Formadores de Mercado Automatizados, em tradução livre). Esse conceito foi criado por Vitalik Buterin em um post intitulado "*On-chain market makers*" (formadores de mercado na rede), e se refere à ideia da criação de um mercado de trocas de ativos baseados em sua liquidez, e não em negociações diretas entre as partes. O Uniswap é uma DEX que permite a troca de *tokens* de diferentes ativos, com base na liquidez, agindo como um ponto de intersecção entre mais de 200 aplicações DeFi.

O Bloom é um DeFi Ethereum que, inicialmente, se propôs a lidar com questões referentes a dados pessoais, servindo como um gerenciador de identidade digital dentro do blockchain. Com o tempo, ele passou a ser utilizado como método de monitoramento da saúde financeira de seus representados, servindo de forma extremamente importante para a geração de *scores* de crédito virtuais, mesmo em âmbitos internacionais. Com o uso dos *tokens* do Bloom o usuário tem a possibilidade não apenas de gerenciar seus próprios dados, mas também garantir a integridade de

informações na blockchain, evitando, por exemplo, a ocorrência de fraudes em seus sistemas.

Existem, ainda, iniciativas desenvolvidas na área de seguros descentralizados. Um projeto importante, o Nexus Mutual, construído na Rede Ethereum, utiliza *tokens* NXM para oferecer o que o próprio projeto chama de “risco compartilhado”. Através de seus *tokens*, a comunidade de usuários gera um fundo financeiro utilizado para proteger seus membros em caso de eventos adversos tanto em seus *smart contracts* e transações envolvendo criptoativos, como o próprio projeto publica ter planos para assegurar danos físicos por eventos naturais como terremotos, por exemplo.

O último projeto que será mencionado aqui é o Augur, um DeFi do Ethereum que se propõe a trabalhar em cima de previsões sobre diversos assuntos, como mercado, esportes e política, e que tem entre seus membros de conselho o próprio Vitalik Buterin. O projeto possui seu próprio *token*, o REP, utilizado por usuários que desejem participar de discussões sobre resultados de algum tópico em aberto na plataforma, apostando valores sobre suas previsões e recebendo recompensas, caso estas estejam corretas.

O DeFi é uma tecnologia nascente, que recém-inicia sua jornada, mas que já apresenta números e resultados surpreendentes. Nos parágrafos anteriores citamos alguns projetos importantes desenvolvidos na Rede Ethereum com ênfase em finanças; a lista, porém, está longe de estar completa, e muitas aplicações não foram sequer citadas. Por exemplo, deixamos de falar de iniciativas de *crowdfunding* (e aqui existe um conceito muito interessante, que vale ser pesquisado, chamado de “contribuições quadráticas”, utilizadas em blockchain), iniciativas em *open banking*, tratativas em mercados de derivativos, entre muitos outros. Isso é ainda reforçado pela ideia de que, nos próximos anos, esperamos testemunhar um aumento não só do número de iniciativas, mas também de classes de soluções financeiras sendo ofertadas com base no blockchain.

4.5 Tokens Não Fungíveis (NFTs)

O mercado dos NFTs é outro que, junto com os DeFis, testemunhou um crescimento muito importante.

Um NFT é, como já visto, um *token* que, diferentemente de um *token* de utilidade, não permite a seu possuidor o usufruto de algum tipo de produto ou serviço, mas o dota de um comprovante de posse, cujo valor pode ser muito alto, por exemplo, em um contexto de mercado de arte ou de artigos colecionáveis. Somente nos três primeiros meses de 2021 a Forbes indica um crescimento de 1785% no valor de mercado de NFTs, e isso em dados preliminares.

Por se tratar de um *token*, residindo em uma blockchain, ele é imutável por natureza e, portanto, uma vez verificado ele serve como uma garantia inequívoca de propriedade, podendo ser vendido pelo seu possuidor em troca de determinada quantidade de moeda ou outro ativo financeiro. Seu valor advém principalmente do conceito da escassez do produto ou serviço a que ele se refere, e seu preço é consequência direta disto. Por não ser fungível, ele não pode ser trocado diretamente por outro *token*.

Uma das primeiras experiências no Ethereum com os NFTs, e base para o desenvolvimento do protocolo ERC-721, os *CryptoPunks* são emblemáticos no que diz respeito a esse tipo de aplicação: trata-se de uma coleção de 10 mil personagens virtuais, diferentes uns dos outros, criados por artistas, e únicos na Rede Ethereum. Por possuírem importância histórica, já que foram pioneiros, e por serem escassos (apenas 10 mil disponíveis), suas propriedades são transacionadas atualmente em cifras acima dos 100 mil dólares, por um único personagem.

NFTs vêm sendo utilizados com sucesso em contextos que trabalham com conteúdos digitais, tais como imagens, vídeos, arquivos de som, entre outros. Obras de arte digitais protegidas por NFTs têm recebido grande parte da atenção da mídia, porém sua aplicação não se limita a isso.

A plataforma brasileira Phonogram.me, por exemplo, é a primeira do tipo a atuar em nosso país. Ela foi criada sobre a Rede Ethereum com o intuito de monetizar obras de artistas e do setor musical no geral. Da mesma forma que um artista recebe direitos autorais sempre que sua obra é reproduzida, por ser o proprietário da obra, um usuário que compre um NFT de determinada gravação passa a ser seu proprietário, tendo direito aí, por exemplo, ao recebimento de *royalties* por sua reprodução. Nos Estados Unidos, o NFT de um disco do DJ 3LAU foi vendido por quase 12 milhões de dólares no início de 2021, o que indica o grande potencial deste mercado.

Um NFT pode não necessariamente representar a obra como um todo; o interessado em "tokenizar" seu produto pode estabelecer condições e propriedades de seu NFT. Isso foi feito pela empresária e cantora brasileira Taynaah Reis, que em 2021 fragmentou uma canção de sua autoria em 100 *tokens*, cada um representando 0,08% dos direitos de reprodução da música, colocados à venda ao preço fixo de 333 dólares — tendo ela recebido um lucro de quase 60 mil reais em 24 horas.

Uma crítica bastante comum feita por pessoas que lidam com NFTs pela primeira vez diz respeito à sua utilidade, considerando que arquivos digitais como imagens ou músicas podem ser copiados facilmente. Para essa crítica, existe uma resposta também comum de que este se trata exatamente do caso de réplicas de obras de arte famosas: réplicas perfeitas podem ser compradas de grandes obras, para serem expostas em casa. Porém, apenas a obra original, certificada e reconhecida por especialistas, é a que tem valor tanto financeiro como de importância histórica.

NFTs têm encontrado aplicações também no desenvolvimento de jogos eletrônicos, usados tanto na compra e venda de itens utilizados nos jogos como customizações de personagens ou objetos utilizados nas interações, como também existem jogos construídos totalmente em cima do conceito do NFT, como o *CryptoKitties*, já mencionado neste livro, em que “gatos virtuais” são

vendidos e negociados, podendo serem usados inclusive para procriarem e gerarem novos animais que são vendidos na rede.

Um jogo criado no Ethereum bastante popular, lançado em 2020, chamado *Decentraland*, conta com um mundo virtual governado por uma DAO, e que utiliza três tipos de *token*, sendo dois deles NFTs: o LAND, que permite comprar propriedades no jogo e o Estate, que representa a porcentagem de território a que o jogador tem a posse. O terceiro *token*, chamado MANA, é um *token* de utilidade utilizado no jogo como moeda de troca para as ações. O poder do NFT é tão grande que proprietários de LAND podem criar comunidades virtuais e monetizá-las para os demais jogadores, auferindo ganhos financeiros reais.

Um tipo de NFT que vem ganhando bastante popularidade é o chamado *token* social. Ele engloba uma categoria de *tokens* que são emitidos por pessoas físicas ou comunidades e que dão a seus possuidores a capacidade de interagirem ou aproveitarem de determinados benefícios, tais como, a participação em determinados grupos de mensagens, ou o acesso a materiais exclusivos de ídolos.

Em alguns casos, pessoas influentes podem chegar a "tokenizar" seu tempo, como foi feito por Reuben Bramanathan, muito conhecido na comunidade blockchain, que disponibilizou algumas de suas horas em forma de *tokens*, transacionados livremente. Qualquer pessoa que compre um *token* destes, por exemplo, tem direito a interagir com ele durante uma hora, válido até 31 de dezembro de 2024, sem restrições e sem qualquer controle sobre a venda deste ativo.

O grande poder dos NFTs, possivelmente, se encontre na intersecção destes com o DeFi. Afinal, um dos benefícios de ser possuir um título de posse é o de aproveitar seu valor para conseguir vantagens.

Aplicações DeFi de realização de empréstimos podem aceitar NFTs como garantia, por exemplo, em troca de melhores taxas, ou como

possibilitadores de crédito. DEXs podem transacionar NFTs fracionados, como é o caso de *tokens* que representam uma porcentagem de direitos de uma obra, ou eles podem ainda serem vendidos diretamente em *marketplaces* de uso geral.

Por último cabe mencionar que até mesmo o ENS apresentado anteriormente se trata de um NFT. Na realidade, quando um leilão é feito sobre a propriedade do registro do nome, o que está sendo transacionado é sua propriedade — esta, assegurada por um NFT.

NFTs são construídos principalmente com base no protocolo ERC-721, porém outros protocolos também estão disponíveis no Ethereum, como o ERC-1155, que cria *tokens* chamados "semifungíveis" e o EIP-2309, que propõe um funcionamento mais otimizado deste tipo de tecnologia.

4.6 Escalabilidade no Ethereum

Anteriormente neste livro mencionamos o fato de o sucesso da Rede Ethereum ter causado alguns inconvenientes para seus usuários, como um maior tempo de processamento de transações e aumento das taxas pagas na forma de *Gas*, ambos consequências da dificuldade de escalabilidade que a própria tecnologia blockchain apresenta.

Em forma geral, espera-se que o Ethereum 2.0 resolva grande parte destas dificuldades. No entanto, até que sua versão seja liberada para o público, outras estratégias precisaram ser criadas de forma a possibilitar o trabalho de usuários e desenvolvedores que utilizem soluções hospedadas na rede.

O grande desafio de se propor uma solução de escalabilidade em blockchain é o de manter a aderência aos requisitos da tecnologia, entre eles — e, talvez, o mais importante —, a questão da descentralização. Diversos projetos e soluções vêm sendo

desenvolvidos e testados, e costumam ser divididos em soluções de escalabilidade “na cadeia” (*on-chain*) — tendo como principal representante a solução de *sharding* (“fragmentação”), em que trechos de informação são separados em pedaços e processados separadamente —, e “fora da cadeia” (*off-chain*) — categoria que inclui as soluções que utilizam cadeias apartadas da *mainnet*.

Outras soluções apresentam uma mistura de características do *on-chain* com o *off-chain*. Enquanto o conceito de *sharding* está diretamente atrelado à publicação do Ethereum 2.0, a seguir discutiremos as principais soluções oferecidas, utilizadas atualmente.

Soluções de escalabilidade *on-chain*

As soluções de escalabilidade *on-chain*, para existirem, precisam, necessariamente, atuar em cima do protocolo do próprio Ethereum sendo, portanto, possíveis de serem executadas apenas pelo time de desenvolvimento do próprio projeto, ou que trabalhem em parceria, apesar de alguns grupos ofertarem soluções deste tipo.

A adoção do *sharding* com o Ethereum 2.0 é um exemplo de solução *on-chain* e, se acordo com a documentação oficial do projeto, é reconhecida também como o principal foco para este tipo de escalabilidade. Sua explicação e funcionamento já foram abordados anteriormente neste livro.

Algumas soluções *on-chain* comerciais incluem o desenvolvimento de máquinas virtuais compatíveis com a EVM e que dizem ser capazes de controlar seu funcionamento, modulando a passagem de informações para a *mainnet* diretamente, o que as classificaria dentro do rol de soluções desta categoria, por atuarem na camada principal.

Soluções de escalabilidade *off-chain*

As soluções de escalabilidade *off-chain*, por sua vez, podem ser divididas em diferentes categorias, dependendo de como elas são aplicadas, tendo em comum o fato de não exigirem qualquer alteração no protocolo Ethereum em uso.

Soluções ditas da “segunda camada” (*layer 2 solutions*) utilizam a estrutura da *mainnet* para garantir a segurança e reaproveitam algumas de suas funcionalidades, como os conceitos utilizados nos protocolos de consenso, enquanto outras consideram a implementação de uma blockchain totalmente apartada, que se limita a manter algum tipo de comunicação com a cadeia principal.

De acordo com a documentação oficial do Ethereum, a maioria das soluções da segunda camada trabalha com base em um servidor (ou um *cluster* de servidores), responsável por processar as transações de forma apartada e, uma vez aprovadas (isso é, após o consenso), são levadas para a *mainnet* — a "camada 1", ou "primeira camada". Incluídas aí, aproveitam-se das características de imutabilidade e segurança da rede.

Uma solução muito comum deste tipo é a que reside no uso de um tipo especial de contrato, chamado de "*multisig*" (do inglês, "*multi-signature*" — “múltiplas assinaturas”). Como seu nome sugere, é um contrato que, para ser executado, necessita ser “assinado” pela inclusão de diferentes chaves primárias — isso é, é validado por mais de um usuário.

Neste modelo, uma transação inicial é responsável por abrir o contrato e manter seu estado, de forma reservada, em localidade apartada da *mainnet* (na camada 2). Os outros participantes, conhecidos e predeterminados, a partir de então ficam livres para enviarem suas transações para ele, utilizando suas assinaturas, até que uma última assinatura o libera, submetendo-o ao processamento na blockchain (apenas então, para a camada 1).

Esse processo age como um “canal” que permite o envio de um grande número de transações à rede de uma única vez,

aumentando o *throughput* de transações, a uma taxa muito menor do que a que seria cobrada caso elas fossem enviadas individualmente. Por esse motivo, suas duas implementações disponíveis são denominadas de “Canais de Estados” (*State Channels*) e “Canais de Pagamentos” (*Payment Channels*), sendo estes especializados na transferência de quantias financeiras entre dois ou mais participantes, e aqueles, de uso geral.

Principal método de construção utilizado em processos de escalabilidade na segunda camada, sugerido por Joseph Poon e Vitalik Buterin, o Plasma foi apresentado em uma publicação de 2017 intitulada “*Plasma: Scalable Autonomous Smart Contracts*” (“Contratos Inteligentes Autônomos Escaláveis”, em tradução livre), como um *framework* desenvolvido para a execução de *smart contracts*, com o potencial de escalar até o nível de bilhões de atualizações de estado por segundo a partir da aplicação dos conceitos de operações de *MapReduce* e aplicação de um mecanismo de consenso baseado em PoS.

Neste método, a segurança dos dados é garantida criptograficamente, por meio das já conhecidas Árvores Merkle (ou suas modificações) — que possibilitam a criação de um grande número de cadeias-filhas baseadas na cadeia principal, capazes de processarem transações de forma independente e escalável. A comunicação com a cadeia principal, por sua vez, é feita através de mecanismos de "Provas de Validade" (*Validity Proofs*) ou "Provas de Fraude" (*Fraud Proofs*), mecanismos de publicidade que permitem aos participantes verificarem a lisura do processo antes de que as transações sejam incluídas à cadeia.

No modelo de *Validity Proofs*, as transações processadas são armazenadas em *batches* e seu estado final é enviado para a primeira camada juntamente com a sua "prova de validade", enquanto o mecanismo das "Provas de Fraude" (*Fraud Proofs*) também trabalha com o conceito de lotes, porém, utiliza um modelo de anotação de transações baseado no UTXO, que valida contabilmente que não existam transações irregulares. Estas são enviadas para a camada 1 por inteiro, mas seu mecanismo de funcionamento permite que os dados sejam "desafiados" quanto à sua validade, quando haja desconfiança, através das "provas de fraude", antes de sua inclusão.

Alguns projetos importantes que utilizam o *Plasma* diretamente são o *Plasma Cash*, utilizado principalmente para o armazenamento e transferência de NFTs, o *Plasma Debit*, com estrutura similar, capaz de trabalhar com NFTs e transações complexas, e o *Minimum Viable Plasma*, utilizado no processamento de pagamentos mais simples.

Soluções que realizam o processamento das transações em uma camada apartada, tal como o sugerido pelo método *Plasma*, são classificadas em um grupo denominado de *rollups* (“rolagens”). Nele, as soluções baseadas em *Validity Proofs* são subclassificadas como pertencentes ao tipo *Zero Knowledge Rollup* - ZK-Rollup, que validam o estado das transações com um mecanismo chamado SNARK (*Succinct Non-interactive Argument of Knowledge*, ou "Argumento de Conhecimento Sucinto Não-Interativo"), enquanto as soluções baseadas nas *Fraud Proofs* são chamadas de *Optimistic Rollups* — ORs. Ambas possuem suas vantagens e desvantagens: uma ZK-Rollup, por exemplo, é mais rápida que uma OR, e pode ser processada em paralelo, aumentando seu grau de descentralização. Uma OR, por sua vez, é mais flexível, e pode ser adaptada a um maior número de casos de uso.

Existem soluções híbridas, que combinam ambos os tipos de rolagens, como é o caso das soluções *Celer* e *Arbitrum*. Outras

soluções, no entanto, são baseadas em um *rollup*, porém apresentam abordagem diferente — o *Validium*, por exemplo, é baseado em um ZK-Rollup, porém, não envia dados para a camada 1.

Uma última solução *off-chain*, bastante popular, sugere a aplicação das chamadas “cadeias laterais” (*sidechains*), que são implementações independentes de blockchains compatíveis com o protocolo Ethereum e que podem, portanto, se comunicar, além de serem capazes de compartilhar contratos. Por serem compatíveis, podem ser entendidas como uma cadeia Ethereum apartada — portanto, com as mesmas capacidades, como a hospedagem de dApps e trabalho com *tokens* construídos com os mesmos padrões.

Sidechains possuem aplicações em diversos contextos, incluindo microtransações (MTX), como pagamentos feitos em jogos eletrônicos, transações estáveis, e transações de aplicações específicas — como aquelas oriundas de um sistema de votação, por exemplo.

Uma cadeia lateral é capaz de funcionar de forma totalmente apartada da *mainnet*, e isso inclui seus processos de segurança e manutenção do consenso. Sua comunicação com a rede principal, normalmente, é feita através de uma “ponte” que permite a comunicação em ambas as direções — também chamado de “comunicação de duas vias” (*two-way peg*).

Um *peg* pode ocorrer de duas formas: simétrica, em que a transação é enviada para a cadeia principal e dela para a lateral através da execução de um protocolo de consenso específico para esta comunicação (havendo uma etapa de sincronização das cadeias), ou assimétrica, em que diferentes protocolos são usados para cada sentido da comunicação.

Soluções comerciais de escalabilidade

Um projeto extremamente importante, iniciado como uma solução de camada 2 e que se consolidou como um meio de integração de blockchains, o *Polygon* surgiu em 2017 sob o nome de *Matic Network* (“Rede Matic”).

Inicialmente trabalhado como um protocolo utilizado para a construção de redes com base na ideia de funcionamento do *Plasma*, atualmente o *Polygon* oferece uma gama de soluções baseadas em uma cadeia lateral, o *Polygon PoS Network*, que trabalha, ela própria, com uma arquitetura baseada em camadas que permitem o processamento distribuído das transações.

O *Polygon* oferece um componente extremamente importante chamado *Polygon Bridge*, capaz de realizar a comunicação entre diferentes blockchains compatíveis com o Ethereum, tais como a já citada Avalanche, que tem ênfase nas soluções para DeFi, permitindo o trabalho no modo chamado *multi-chain* (múltiplas cadeias) — que possibilita o aproveitamento das vantagens de cada uma.

Outros projetos, que também trabalham na ideia de integração de cadeias de blockchains e que precisam ser citados são o *Polkadot* e o *Cosmos*.

O *Polkadot* foi lançado em 2020 como uma rede que utiliza o consenso baseado em PoA e oferece a hospedagem em uma cadeia principal chamada *Relay Chain* (Cadeia de Revezamento), sobre a qual são criadas cadeias específicas para realizar a computação de transações — as *Parachains*.

O *Cosmos*, por sua vez, possui arquitetura bastante similar. Ele oferece uma cadeia principal chamada *Hub*, conectada a outras cadeias, denominadas *Zones* (“zonas”). A comunicação entre cada zona é feita através de um protocolo, denominado *Inter-Blockchain Communication* (“Comunicação Entre Blockchains”), IBC, utilizado para a troca de mensagens.

Tanto o *Polkadot* como o *Cosmos* também possuem mecanismos de comunicação com outras blockchains, através de *bridges* específicos, existindo um balanço de vantagens e desvantagens entre eles.

Conclusão

Com a conclusão deste capítulo, encerramos a primeira parte deste livro, em que não apenas apresentamos a blockchain como tecnologia, mas em que também introduzimos os principais fundamentos de funcionamento do Ethereum e suas aplicações.

Com o conhecimento adquirido até agora, você já é capaz de participar de discussões não apenas técnicas, mas também que envolvam decisões estratégicas sobre o uso da blockchain — além de conseguir fazer a leitura crítica sobre artigos e apresentações que citem sua aplicação.

A partir de agora nosso foco será um pouco mais prático: conhecendo os modos de operação do Ethereum, começaremos a construir aplicações que utilizem sua estrutura como base para a execução de programas e, para isso, é fundamental o bom entendimento de tudo o que foi exposto até aqui, visto que as principais decisões sobre design e implementação que serão vistas nos próximos capítulos são consequências diretas de sua interação com o *core* da blockchain.

Parte 2 - DESENVOLVENDO PARA A BLOCKCHAIN ETHEREUM

Capítulo 5
Preparando o ambiente de desenvolvimento

5.1 Introdução

Antes de iniciarmos o estudo da linguagem de programação Solidity e fazermos as implementações de nossos primeiros *smart contracts* e seus respectivos testes, precisamos ter um ambiente de desenvolvimento preparado, configurado e funcional.

Em um cenário real, a simples publicação e realização de testes em nossos contratos inteligentes demandaria o pagamento de taxas - principalmente na forma de *Gas*, visto serem etapas que envolvem tanto a criação de transações como sua inclusão em blocos dentro do Ethereum. Para isso, teríamos, também, que possuir certa quantidade de Ether disponível para ser usada (o que pode trazer alguns prejuízos financeiros em um cenário de múltiplas execuções para fins de garantia de qualidade).

Para essas situações, temos algumas ferramentas de teste, que nos proveem um ambiente que simula, com bastante realismo, o funcionamento de uma blockchain localmente, em que utilizamos moedas sem valor real e que podem ser facilmente recriadas e utilizadas.

Nas próximas seções veremos passo a passo a preparação do ambiente de desenvolvimento que será usado. É importante lembrar que este não é um livro com foco no estudo de frameworks

específicos e que, portanto, nos limitaremos a mostrar o funcionamento da linguagem, em alto nível.

Um framework extremamente popular para o desenvolvimento de dApps é o *Truffle.js*, e fica como sugestão de estudos para a leitora ou leitor que tiver interesse em se aprofundar tecnicamente no assunto.

5.2 Node.js

Iniciaremos nosso *setup* com a instalação do Node.js, um ambiente de execução *server-side* para o JavaScript capaz de executar trechos de códigos desenvolvidos nesta linguagem de programação sem a necessidade de um cliente, tal como um *browser*, por exemplo. É uma solução *open source*, multiplataforma, que permite a execução de programas em uma variedade de sistemas operacionais, tal como o Windows, Linux ou macOS.

O Node.js é um *runtime* JavaScript assíncrono, baseado em eventos, que permite a construção de aplicações bastante escaláveis, capaz de gerenciar uma série de conexões de forma concorrente.

Apesar de não necessariamente trabalharmos diretamente com o JavaScript no desenvolvimento de nossos contratos, vamos nos aproveitar do *Node Package Manager* (NPM) — o gerenciador de pacotes do Node.js — para instalar algumas entidades externas, tais como o compilador Solidity (o compilador da linguagem de programação que utilizaremos), de forma bastante facilitada.

Como já teremos um ambiente de desenvolvimento JavaScript pronto, aproveitaremos para exemplificar o uso do web3.js, a API JavaScript do Ethereum, que, apesar de não necessária para o desenvolvimento com Solidity, é bastante utilizada em projetos reais por permitir a comunicação entre uma linguagem de programação

popular - o JavaScript - com nossos *smart contracts*, o que não apenas facilita muitas tarefas, mas também permite a criação de níveis de abstração bastante robustos.

A instalação do Node.js pode ser feita diretamente do site oficial do projeto, no endereço https://nodejs.org/en/download/, pela seleção do sistema operacional em uso. No momento da publicação deste livro, o Node.js está em sua versão 16.13. Qualquer versão mais recente pode ser utilizada sem prejuízos para nosso experimento. Deixaremos o passo a passo de instalação por sua conta; em caso de dúvidas, consulte a documentação oficial do Node.js.

Ao finalizar a instalação do ambiente de execução, o usuário acaba instalando o NPM como parte da solução. Ele é, atualmente, o maior repositório online de projetos *open source* desenvolvidos com o Node.js disponível na rede, e ele também disponibiliza uma interface via linha de comando para sua interação, permitindo realizar a instalação e atualização de pacotes, bem como a solução de dependências e versionamentos, que utilizaremos nas próximas seções.

5.3 Compilador de pacotes C/C++

Alguns dos pacotes de desenvolvimento Ethereum que vamos utilizar são instalados através do NPM, como já citado. Porém, muitos deles utilizam compiladores de pacotes C/C++ em tempo de execução, o que torna necessária sua instalação em forma separada.

Usuários do sistema operacional Linux e suas distribuições e do macOS podem seguir com a instalação destas bibliotecas de forma bastante simplificada, através de comandos de instalação dos metapacotes *build-essential*, que incluem uma série de dependências instaladas automaticamente, tais como compiladores

e bibliotecas GCC/g++, ferramentas de desenvolvimento e *debug GNU* e o *Make*, utilitário de compilação automatizado utilizado desde a década de 1970 e que, até o momento, se mantém bastante popular.

No caso do Linux e do macOS, a instalação do *build-essential* pode ser feita diretamente via linha de comando, como nos seguintes comandos, utilizados para o Ubuntu:

```
sudo apt-get update

sudo apt-get install build-essential
```

Caso você esteja trabalhando com outras distribuições que não o Ubuntu, você pode buscar por maiores informações sobre instalação do *build-essential* em motores de busca específicos ou em fóruns para as diferentes distribuições, nos quais a comunidade Linux costuma ser muito prestativa.

Usuários Windows necessitam de um pacote de ferramentas chamado *Windows Build Tools*, cuja instalação, no Windows 10, pode ser feita diretamente a partir do *Windows Power Shell*, aberto com permissão de administrador. Sua instalação é feita pelo próprio NPM, pela digitação do comando:

```
npm install –global –production windows-build-tools
```

5.4 Inicializando o projeto via NPM

Deixamos a cargo da leitora ou do leitor definir o local em que deseja criar o projeto do *smart contract*, bem como definir a melhor estrutura de pastas que se adeque às suas necessidades.

Aqui, apenas orientamos a que, uma vez a estrutura tenha sido criada, o projeto em Node.js seja inicializado a partir da digitação do seguinte comando, no terminal:

```
npm init
```

Esse comando é utilizado no NPM para inicializar projetos, no Node.js ou, mais corretamente, inicializar um novo pacote NPM. Seu efeito imediato é a criação do arquivo `package.json`, que contém os metadados do projeto, tais como nome, versão, relação de *scripts* a serem executados quando o projeto é inicializado, dependências, entre outros.

Ao digitar o comando no terminal, o NPM solicitará uma série de informações que serão utilizadas e armazenadas no arquivo, de preenchimento recomendado, porém, opcional, como o nome do pacote que estamos criando, sua versão, uma descrição do que está sendo feito, a identificação do ponto de entrada de nosso código, comandos para execução de teste, endereço de repositório Git, uma lista de palavras-chave, entre outras. Caso não desejemos dar alguma destas informações (ou não queiramos preencher nenhuma delas), podemos simplesmente deixar em branco (neste caso os valores utilizados são *default*) e ir para a próxima linha.

5.5 Chalk

A biblioteca *Chalk*, disponibilizada pelo NPM, é uma dependência que facilita a escrita de mensagens no console do Node.js durante a execução dos projetos, permitindo a personalização, por exemplo, para a inclusão de cores de destaque. Ela é útil nas fases de desenvolvimento, para conseguirmos visualizar parâmetros de resposta, objetos criados em tempo de execução, entre outros e será usada, aqui, de forma meramente didática. Esta instalação não é obrigatória para nossa implementação.

Para visualizar as versões disponíveis da biblioteca, podemos usar o comando:

```
npm show chalk versions
```

No momento de escrita deste livro, a versão mais atualizada disponível é a 5.0.0. Qualquer versão mais recente do Chalk pode ser utilizada, podendo ser instalada via terminal, estando no diretório do projeto, pela digitação de um dos seguintes comandos, que, respectivamente, instalam a última versão disponível ou instalam uma versão específica (no caso, a 4.1.1, por exemplo).

```
npm install chalk --save-dev

npm install chalk@4.1.1 --save-dev
```

O sufixo `--save-dev` é incluído aqui para que o Chalk seja entendido pelo projeto como uma dependência de desenvolvimento, não sendo necessário para a execução do nosso projeto, uma vez que ele seja publicado.

5.6 Mocha

O *Mocha* é um framework para desenvolvimento e execução de testes em JavaScript, nativo do Node.js, sendo capaz também de ser executado via *browser*. Seus testes são executados em série, com a possibilidade de geração de relatórios e mapeamento de exceções, o que o torna uma ferramenta bastante versátil e útil. É um framework extremamente popular, utilizado por desenvolvedores em diversos casos de uso de aplicações.

As versões disponíveis do Mocha podem ser consultadas pela digitação do seguinte comando, via terminal:

```
npm show mocha versions
```

Neste momento, a versão mais recente da biblioteca é a 9.1.3. Sua instalação pode ser feita, tal como exemplificado anteriormente, digitando-se o nome do pacote, seguido da anotação do símbolo de `@` e, então, a versão específica de trabalho desejada. Caso essa informação não seja incluída, a versão mais atual será instalada.

Ambas as versões do comando são mostradas a seguir (no caso do comando com versão, estamos considerando a versão 9.1.3), e incluem o comando `--save-dev` que, como já mencionado, inclui o pacote como uma dependência utilizada exclusivamente no ambiente de desenvolvimento. Sua execução deve ser feita estando no diretório do projeto, via terminal:

```
npm install mocha --save-dev

npm install mocha@9.1.3 --save-dev
```

5.7 Compilador Solidity

Existem diferentes compiladores utilizados para o Solidity, que possuem a mesma função básica: converter o código em um *bytecode*, que é interpretado pela EVM, executando o programa.

Em nosso laboratório trabalharemos com o *solc*, compilador recomendado pela documentação do projeto Solidity. Em sua instalação via NPM trabalharemos com o *solcjs* que, além do compilador, traz alguns *bindings* específicos para o trabalho com JavaScript. Tanto o *solcjs* como o *solc* são projetos *open source*, e seus repositórios Git estão disponíveis. Seus endereços são citados nas referências, no capítulo final.

Para visualizar as versões disponíveis do compilador no NPM, o seguinte comando pode ser digitado a partir do terminal:

```
npm show solc versions
```

No momento que este livro está sendo escrito, a versão mais recente do compilador é a 0.8.x, que já inclui suporte à atualização *London*, publicada em agosto de 2021. Para nosso experimento, qualquer versão atual deveria funcionar sem maiores intercorrências, porém, visto que o Solidity lança novas versões em curtos espaços de tempo e que sua especificação é alterada

constantemente, recomendamos trabalhar com a versão 0.8.12, que é a mesma que utilizaremos neste livro. Isso pode ser feito através da digitação do comando:

```
npm install solc@0.8.12 --save-dev
```

Reforçamos que o sufixo `--save-dev` é usado pelo NPM para informar que o pacote será utilizado exclusivamente durante nosso desenvolvimento, não sendo necessário para a execução do pacote NPM sendo criado (no caso, o nosso projeto), uma vez que ele esteja pronto e tenha sido publicado.

5.8 Editor de código e plugin Solidity

Em nosso laboratório utilizaremos o *Visual Studio Code* (*VS Code*) como ambiente de desenvolvimento. Trata-se de um editor de código bastante popular, *open source*, extremamente versátil, com suporte a um grande número de extensões de suporte de linguagem e *runtimes* para execução. Além disto, ele pode ser executado tanto no Windows como em distribuições Linux e no macOS, o que o torna a ferramenta de trabalho ideal para nossa proposta.

A instalação do VS Code pode ser feita diretamente a partir de seu site, no endereço https://code.visualstudio.com/, pela seleção o sistema operacional de uso.

Aqui, qualquer versão atual disponível do programa pode ser utilizada, sem prejuízos para a nossa execução. Deixamos com você a missão de seguir com o passo a passo da instalação, de acordo com seu sistema operacional, a partir da própria documentação disponível no site do programa.

Em seguida, recomendamos instalar algum *plugin* específico para o Solidity dentro do *marketplace* do VS Code. Apesar de não ser uma etapa obrigatória, ela auxilia muito no desenvolvimento ao fornecer,

por exemplo, suportes à compilação e execução de realces de sintaxe (*syntax highlighting*).

Particularmente, utilizamos o *plugin* Solidity desenvolvido por Juan Blanco, que é bastante popular e traz funcionalidades que vão além do *syntax highlighting* e da criação de *snippets*, chegando a disponibilizar modelos de protocolos ERC utilizados para a geração de *tokens*, e geração de código para o *Nethereum*, utilizado no desenvolvimento de soluções para o Ethereum no .NET.

5.9 Web3.js

O web3.js (comumente abreviado para “web3”) é uma biblioteca que implementa a chamada API Ethereum do JavaScript. É um módulo extremamente popular e, provavelmente, a mais madura das bibliotecas disponíveis para o trabalho com Ethereum utilizando componentes externos ao Solidity. Segundo a documentação oficial do Ethereum, ela é utilizada tanto no desenvolvimento *front-end* como no desenvolvimento *back-end*, para realizar leituras diretamente da blockchain ou durante a criação e execução de transações.

A maioria das funções implementadas pelo web3 é assíncrona; isto, devido à própria natureza dos protocolos de comunicação do Ethereum, em que a comunicação com os nós é feita através de chamadas JSON RPC. É muito importante ter atenção a esse fato durante a implementação de suas chamadas para evitar exceções devidas à indisponibilidade de dados em determinados momentos.

As versões disponíveis do web3 podem ser consultadas pela digitação do seguinte comando, via terminal:

```
npm show web3 versions
```

No momento da publicação, a versão disponível mais recente da biblioteca é a 3.0.0-rc.5. Sua instalação pode ser feita, tal como exemplificado nos itens anteriores, digitando-se uma versão específica de trabalho, pela inclusão de seu número após o símbolo de `@`, durante o comando de instalação, ou sem a informar — o que terminará por instalar a versão mais atual naquele momento (que possivelmente possa já estar atualizada em relação ao nosso livro).

Estando no diretório do projeto, no terminal, os comandos para sua instalação, sem indicar uma versão específica e, indicando a versão 3.0.0-rc.5, por exemplo, são, respectivamente:

```
npm install web3 --save-dev

npm install web3@3.0.0-rc.5 --save-dev
```

Conforme já citado, os parâmetros `--save-dev` indicados ao final dos comandos indicam que sua utilização como dependência deve ser considerada exclusivamente dentro do ambiente de desenvolvimento.

5.10 Ganache

O *Ganache* é uma funcionalidade do *Truffle Suite* que provê uma blockchain Ethereum “de uso pessoal”, extremamente popular e útil para as diferentes etapas de desenvolvimento de nossos *smart contracts*, apoiando tanto a etapa de codificação como a publicação e realização de testes em dApps, de maneira segura. É uma *testnet* classificada como privada, por ser de uso exclusivo do desenvolvedor, que a roda localmente.

Ao ser executado, o Ganache simula o funcionamento de uma blockchain, hospedada na máquina local, e também provê um número determinado de contas — cada uma com uma quantia de criptomoedas, independentes. Suas versões estão disponíveis para

plataformas Windows, Linux e macOS, e são também úteis para o desenvolvimento de outras blockchains, como projetos baseados no *Corda*, por exemplo.

Sua instalação pode ser feita diretamente do site do projeto, https://www.trufflesuite.com/ganache, através da escolha do sistema operacional em utilização, ou pela visita direta a seu repositório — https://github.com/trufflesuite/ganache/releases.

Tal como fizemos com as outras aplicações, você deve seguir o passo a passo necessário para as instalações, de acordo com a documentação oficial disponibilizada ali mesmo.

Ao ser executado, o projeto mostra alguns parâmetros sendo utilizados pela blockchain em simulação, tais como o número do bloco atual e limite do *Gas*. Também dá uma lista contendo dez contas representados por endereços lógicos mostrados como um hexadecimal, cada uma com uma quantidade inicial fictícia de 100.00 ETH disponíveis para uso.

Conforme seu uso, esses parâmetros vão sendo atualizados em tempo real, exatamente como ocorreria em um cenário real de utilização de uma blockchain, o que é muito útil para a realização de nossos testes, além de ser uma ferramenta didática valiosa.

Quando as transações são incluídas, observamos os blocos sendo minerados e incluídos à cadeia, os valores financeiros sendo movimentados entre as contas, e temos acesso a detalhes de transações, contratos e eventos disparados na rede. Isso ficará claro em capítulos posteriores, quando fizermos a implementação de nosso contrato.

Caso alguma das contas fique sem dinheiro para ser movimentada, podemos facilmente reinicializar a aplicação, retornando-a ao estado inicial, sem prejuízos financeiros reais.

5.11 Setup inicial do projeto

Caso todos os passos anteriores tenham sido seguidos corretamente, devemos ser capazes de ver as dependências listadas corretamente dentro do arquivo `package.json`, dentro do item `devDependencies` em nosso projeto (que pode ser consultado tanto via linha de comando como diretamente pelo uso do VS Code), que indicam as dependências que são consideradas apenas no ambiente de desenvolvimento.

Além de termos instalado o *solc* em nosso projeto, incluído como uma dependência, precisamos configurar nossa IDE para utilizá-lo durante as etapas de compilação. Isso é realizado através das configurações de preferências do VS Code, através da navegação menu *File* > *Preferences* > *Settings*.

Dentro da aba de *Workspace Settings*, selecionamos a opção *Extensions*, e procuramos pela opção *Solidity Configuration*. Dentro do quadro *Compile Using Remote Version*, indicamos a versão do compilador que será utilizada para execução do projeto — a mesma versão que aparece no arquivo `package.json`. Caso isso não seja feito, o VS Code trabalhará com a versão mais atual do compilador e, eventualmente, isso pode gerar problemas de compatibilidade.

Ao alterar a configuração, aparecerá no menu lateral do VS Code um arquivo `settings.json`, dentro da pasta `.vscode`, contendo um JSON, que aponta para a utilização desta versão do compilador. Neste momento, o VS Code precisa ser reinicializado. Após a reinicialização da IDE podemos prosseguir com o setup inicial do projeto.

A partir da raiz do projeto, criamos uma pasta chamada `contracts` e, dentro dela, um arquivo chamado `TestContract.sol`, que será utilizado simplesmente para testarmos se nosso ambiente está configurado e funcionando corretamente.

Sem entrar em detalhes, já que o funcionamento do Solidity será visto mais à frente, a primeira instrução de um arquivo de um *smart contract* deve ser a declaração da licença sobre a qual o código está sendo criado, e esta declaração é feita na forma de um comentário; aqui, trabalharemos com a licença no padrão MIT. Em seguida, fazemos a declaração do compilador que está sendo utilizado e sua versão.

A informação sobre o compilador é dada dentro de uma declaração especial, chamada *pragma*, indicando a seguir a versão, que determina como o código será tratado ao ser processado pelo compilador. No nosso caso, já que estamos com a versão 0.8.12 do *solc*, a declaração é:

```
// SPDX-License-Identifier: MIT
pragma solidity = 0.8.12;
```

Em seguida, fazemos a declaração do contrato, feita utilizando a palavra reservada `contract`, seguida pelo seu nome, e a abertura e fechamento das chaves. Ao final, a estrutura de declarações do nosso arquivo é semelhante ao trecho a seguir:

```
// SPDX-License-Identifier: MIT
pragma solidity = 0.8.12;

contract TestContract{}
```

Neste momento, passamos à criação do arquivo de compilação do nosso contrato. Na raiz do projeto, criamos uma pasta, chamada `scripts` e, dentro dela, criamos o arquivo `compile.js`.

Inicialmente, no arquivo, importamos uma série de bibliotecas JavaScript, bem como o compilador. Em seguida, localizamos o arquivo contendo o código-fonte de nosso contrato de teste utilizando as bibliotecas importadas *path* e *fs*, que permitem trabalhar, respectivamente, com caminhos dentro da estrutura do projeto e com o sistema de arquivos (*file system* — e daí a abreviação “fs”) a partir do Node.js, tal como demonstrado a seguir.

```
const path = require('path');

const fs = require('fs');

const solc = require('solc');

const contractName = 'TestContract.sol';

const chalk = require('chalk');

const contractPath = path.resolve(__dirname,
                                  '../contracts',
                                  contractName);

const source = fs.readFileSync(contractPath, 'utf8');
```

Nesse ponto, devemos criar um objeto de inicialização do compilador seguindo o modelo disponível na documentação do *solc*, indicando informações tais como a linguagem de programação utilizada, os contratos que estão sendo compilados, seu conteúdo, entre outros. Este é um exemplo de informação que varia muito de acordo com a versão do compilador sendo utilizado; portanto, a importância de estarmos trabalhando todos na versão 0.8.12.

Um exemplo do objeto criado, contendo o mínimo de declarações necessárias para seu funcionamento, é dado por:

```
const compilerInput = {
    language: 'Solidity',
    sources:
    {
        'TestContract.sol':
        {
            content: source
        }
    },
    settings:
    {
        optimizer:
        {
```

```
            enabled: true
        },
        outputSelection:
        {
            '*':{
                '*':['*']
            }
        },
    }
};
```

Aqui nos limitamos a informar apenas as informações obrigatórias; existe uma série de parâmetros opcionais que podem ser utilizados para a realização de ajustes finos ou para expor preferências a serem utilizadas durante o processo de compilação. Informações gerais podem ser consultadas diretamente na documentação oficial do projeto.

A compilação do objeto é feita através de métodos específicos do solc. No trecho de código a seguir, ela é feita através do método 'compile' do solc, passando como parâmetro o objeto `compilerInput`, por exemplo.

Caso ela ocorra com êxito, a invocação do método retorna uma cadeia de caracteres contendo informações relevantes, tais como os metadados da operação, informações sobre armazenamento, informações sobre eventuais erros e o *bytecode* gerado no processo. Esses dados podem ser parseados para melhor visualização e também podem ser impressos diretamente no console.

Por exemplo, mostramos aqui as estimativas de uso de *Gas* para publicação de nosso contrato e também o *bytecode* do contrato compilado, definidos nas constantes `gasEstimates` e `bytecode`. Usamos a biblioteca *chalk* para fazer a impressão no console usando diferentes cores, com fins meramente didáticos.

```
const inputString = JSON.stringify(compilerInput);
```

```
const output = JSON.parse(solc.compile(inputString));



const contractEVM = output.contracts[contractName]
                                    ['TestContract']
                                    ['evm'];


const bytecode = contractEVM['bytecode']
                            ['object'];


const gasEstimateTotal = contractEVM['gasEstimates']
                                    ['creation']
                                    ['totalCost'];


const gasEstimateExecution = contractEVM['gasEstimates']
                                        ['creation']
                                        ['executionCost'];


console.log(chalk.yellow('Total Gas Execution:'));

console.log(chalk.green(gasEstimateExecution))

console.log(chalk.yellow ('Total Gas Estimate:'));

console.log(chalk.green(gasEstimateTotal));

console.log(chalk.yellow('ByteCode:'));

console.log(chalk.green(bytecode));
```

Ao executar o arquivo de compilação via terminal, diretamente no VS Code, vemos os resultados impressos console, indicativos de sucesso do setup.

```
node .\scripts\compile.js
```

O *output* da execução é o seguinte:

```
    Execution Gas Cost:
    66

    Total Gas Cost:
    12666

    Bytecode:
60806040523480156 00f57600080fd5b50603f80601d6000396000f3fe6080604052600080
fdfea2646970667358221220ce60f84a65cfae474debe67daaa5210d0ce5053d1725958715
b488f71349980f64736f6c63430007050033
```

5.12 Uma alternativa rápida: Remix

Como vimos, a preparação de um ambiente para implementação de um contrato baseada em Solidity não é complexa, porém, chega a ser trabalhosa, tendo em vista a quantidade de ferramentas e dependências a serem instaladas. Isto, considerando nosso caso, em que desenvolveremos um microprojeto, com mera intenção de ilustrar a codificação de um projeto baseado em blockchain, sem grandes necessidades ou definições de arquitetura, por exemplo.

Uma alternativa bastante madura utilizada para o desenvolvimento de pequenos trechos de código e que não exige a instalação de quaisquer programas e/ou dependências é o projeto *Remix* — uma IDE que permite o desenvolvimento de *smart contracts* para blockchains baseadas no Ethereum, que pode ser acessada e executada online, pelo endereço https://remix-project.org/, ou utilizada como uma aplicação desktop.

O *Remix* é uma IDE *open source*, escrita em JavaScript, com arquitetura baseada em *plugins*, composta por módulos para

desenvolvimento, *debugging* e publicação de contratos em Solidity. Dentre seus *plugins*, ela oferece suporte para o *Optimism*, utilizado no desenvolvimento de *sidechains* do Ethereum, sendo uma ferramenta extremamente completa e fiel à realidade de desenvolvedores blockchain.

Uma vez acessado, seu ambiente pode ser configurado e personalizado a partir do menu de navegação, dividido em configurações de arquivos de desenvolvimento, configurações do compilador — com opções para versão do compilador, linguagem utilizada e a versão da EVM, entre outras —, opções para a publicação e execução de transações — como seleção de versões da cadeia baseadas em *forks* do Ethereum, definições personalizadas de limite e valor de *Gas* sendo utilizados e, ainda, com uma série de contas fictícias, representadas por endereços lógicos, cada uma com uma quantia inicial de Ether disponível para uso nas simulações — e opções para o uso de *plugins* adicionais.

O *Remix* também fornece alguns arquivos iniciais com exemplos de código utilizados tanto para a geração de contratos, utilização de *scripts* em JavaScript para publicação, e realização de testes unitários, sendo uma alternativa muito popular para uma escrita e uma execução rápida de *smart contracts* sem a necessidade da realização do *setup* do ambiente.

Caso você opte por utilizar o *Remix* como ferramenta de trabalho, recomendamos o estudo de sua documentação para entender seu uso de forma mais profunda.

Conclusão

Tendo concluído o passo a passo para a preparação do nosso ambiente de desenvolvimento estamos prontos, a partir de agora, para iniciar os estudos da linguagem de programação Solidity e realizar a implementação de nossos primeiros *smart contracts*!

Como já citado aqui, o setup do projeto não é complexo, porém, é uma etapa bastante trabalhosa (e cada um de seus passos é relevante, como ficará claro nos próximos capítulos). A estrutura apresentada é útil e madura o suficiente inclusive para a criação de *smart contracts* em cenários reais, com altos graus de complexidade e necessidades e, certamente, poderá ser consultada diversas vezes como um guia para a geração de projetos futuros.

Recomendamos, por fim, que o leitor se familiarize bastante com o funcionamento do *Ganache*, pela realização de testes e pela exploração de suas funcionalidades, que mostrarão de forma muito realista o funcionamento da blockchain, o que ilustra de forma extremamente positiva o que já havia sido apresentado nos primeiros capítulos deste livro.

Capítulo 6
Linguagem de programação Solidity

6.1 Introdução

Iniciamos agora a parte do livro que, sem dúvidas, é a que exigirá maior participação por parte de você que está lendo. Aqui, começamos a abandonar os aspectos puramente teóricos (ainda que não os deixemos totalmente de lado neste capítulo), e começamos a implementação prática de uma aplicação desenvolvida para a blockchain — e, portanto, distribuída, por sua própria natureza.

Nas próximas seções conheceremos a linguagem Solidity. Veremos detalhes de seu funcionamento, os tipos de dados suportados por ela, seus principais operadores, contextos, modificadores, eventos, entre outras funcionalidades. Isto, em preparação para a implementação de diferentes contratos, com graus de complexidade variados, mais à frente.

Por se tratar de um livro de fundamentos, trabalharemos aqui com a linguagem “pura”, sem a utilização de *frameworks* ou extensões. Caso haja interesse em se aprofundar no assunto após a conclusão deste capítulo, recomendamos a busca por maiores informações sobre o *framework Truffle.js*, sem dúvidas o principal utilizado por desenvolvedores Solidity para a criação de aplicações descentralizadas.

6.2 Características do Solidity

O Solidity é uma das linguagens de programação de alto nível reconhecidas pelo time de desenvolvimento do projeto Ethereum,

utilizada principalmente para a criação de *smart contracts* publicados na rede. Damos ênfase aqui a seu uso para a *criação* de contratos, mas não para sua *manutenção*. Isso porque, devido às próprias características da blockchain, uma vez que os contratos tenham sido publicados, eles não podem ser modificados diretamente.

Sua documentação oficial (citada nas referências, no capítulo final) a define como uma linguagem de alto nível, orientada a objetos, influenciada pelo C++, Python e pelo JavaScript. É também uma linguagem tipada estaticamente, com suporte à criação de tipos definidos pelo usuário, e capaz de trabalhar com herança (sendo possível a aplicação do conceito de herança múltipla).

Seu desenvolvimento pode ser modularizado pelo uso de bibliotecas, que diminuem a complexidade do código e sua gramática prevê, ainda, o uso de expressões em linha para a execução de trechos de código em baixo nível, como aqueles executados via Yul e Yul+ que, como visto em capítulos anteriores, são linguagens que permitem o acesso ao *Ewasm* — normalmente usadas para tarefas que exijam um nível adicional de segurança ou maior performance.

O Solidity evolui muito rapidamente e, portanto, sua documentação deve ser consultada constantemente. Em linhas gerais, a equipe de desenvolvimento afirma publicar novas versões a cada duas ou três semanas, sendo que podemos esperar cerca de duas grandes mudanças de versão no ano, com importantes alterações na linguagem. Isso também se deve ao fato de que o Ethereum está evoluindo, o que se torna um desafio importante, visto a necessidade de seus contratos estarem em conformidade com as novas diretrizes.

6.3 Tipos

Por ser uma linguagem definida como estaticamente tipada, a definição dos tipos utilizados pelas variáveis, funções e contratos é tarefa fundamental da pessoa desenvolvera. E aqui entendemos um "tipo" como sendo uma categoria abstrata, representativa do elemento em questão, que o permite agrupar junto a seus elementos semelhantes (de mesma categoria).

Tipos estão associados a diversos elementos dentro do Solidity, e incluem tanto os tipos próprios das variáveis como os dos contratos — que são do tipo *contract* — e funções — do tipo *function*.

O Solidity trabalha com três tipos principais de dados: os Tipos de Valor (*Value Types*), Tipos de Referência (*Reference Types*) e os Tipos de Mapeamento (*Mapping Types*).

Value Types

Value Types são aqueles que, conforme seu próprio nome sugere, são passados por valor entre variáveis — seu valor é "copiado" para outra variável. Fazem parte deste grupo, no Solidity, os tipos:

1. Booleanos (*bool*) — que trabalham exclusivamente com valores verdadeiros (`true`) ou falsos (`false`);

2. Inteiros — estes se subdividem em inteiros com sinal (`int`) e inteiros sem sinal (`uint`), e podem ocupar até 256 bits. Na declaração deste tipo, o Solidity permite a definição do tamanho, variando do `int8` ao `int256` (no caso de inteiros com sinal) e do `uint8` ao `uint256` (no caso de inteiros sem sinal), variando em múltiplos de 8. Caso seu tamanho não seja especificado explicitamente, o compilador entende que está sendo usado o valor de 256 bits;

3. Números de ponto fixo — são utilizados por números com representação decimal. Assim como com os inteiros, são subdivididos em números de ponto fixo com sinal (`fixed`) e sem sinal (`ufixed`) e também podem ter seus tamanhos definidos em sua declaração, na forma `(u)fixedMxN` , em que *M* representa o

número de bits tomados do número inteiro, variando entre 8 e 256 por múltiplos de 8, e *N* representa o número de pontos decimais considerados, variando entre 0 e 80. Caso estes não sejam declarados explicitamente, o compilador entende que estão sendo usados os valores `ufixed128x18` e `fixed128x18`, respectivamente, para os números de ponto fixo sem sinal e com sinal. Números de ponto fixo diferenciam-se dos números de ponto flutuante (tais como o `float` ou o `double` existentes, por exemplo, no C) pelo modo de sua representação. Aqui, eles são baseados em uma parte para representar seu valor inteiro, e outra para sua versão de fração, enquanto números de ponto flutuante trabalham com a representação em expoente e mantissa. A representação como ponto fixo traz importantes ganhos de performance na realização de operações.

4. Endereços (*address*) — não são tipos comumente encontrados em linguagens de programação, e são representações lógicas dos endereços do Ethereum — como já vimos, calculados a partir das chaves públicas das carteiras. Possuem tamanho fixo de 20 bytes, e são utilizados para representar as localidades para as quais o Ether pode ser transferido (em unidades de wei). Seu uso dá acesso a propriedades intrínsecas, como o `balance`, que verifica o saldo de um endereço, útil em etapas de verificação. Uma variável de tipo *address* pode, ainda, ser alterada pelo modificador `payable`, que possibilita o uso de funções especiais (chamadas aqui de *members*), diretamente, tais como o `transfer` (usado para transferir divisas entre uma conta e outra em implementação de alto nível) e o `send` (uma implementação de baixo nível do `transfer`, diferenciando-se dela por sua incapacidade de tratar exceções).

5. Vetores de bytes de tamanho fixo — representam bytes com tamanho até 32, e seu tamanho deve ser declarado no formato `bytesX` em que *X* representa o tamanho desejado (entre 1 e 32). Caso o valor de *X* não seja declarado, entende-se tratar de um vetor de byte de tamanho dinâmico — um tipo de referência, e

não de valor. Uma variável deste tipo é utilizada para operações que necessitem acessar diretamente os bytes para processamento.

6. Enumerações (*enum*) — enumerações representam tipos predefinidos pelo usuário, indicando uma lista de possíveis valores para uma variável deste tipo. São normalmente utilizados para representar estados ou valores possíveis (por exemplo, dias da semana — em que existem apenas sete valores possíveis). Enumerações podem ser convertidas para números inteiros, porém, o contrário não é verdade.

7. Literais — são representações de valores fixos utilizadas no código e, como tal, devem obedecer às regras e expectativas do tipo a que estão representando, e possuem possibilidades de representação. Por exemplo, um literal de endereço deve, obrigatoriamente, possuir 20 bytes (tal como um *address*), além de passar pela prova do `checksum`, que garante ser um endereço válido. Literais do tipo *String* (*String literals*), isto é, cadeias de caracteres ou, de forma simplificada, “textos”, são também consideradas tipos de valor, ainda que a *String* propriamente dita, na realidade, seja entendida como um vetor e, portanto, considerada como um tipo de referência. *String literals* devem ser escritas entre aspas duplas (aspas simples também funcionam, porém, não são recomendadas pelo *styleguide* da linguagem). O Solidity ainda conta com suporte para literais *unicode*, que permitem a inclusão de representações especiais, tais como *emojis*.

Reference Types

Reference Types são tipos cujo valor é passado entre variáveis por meio de sua referência de localização. Enquanto tipos de valor geram uma cópia do elemento original, tipos de referência possuem

localização "compartilhada"; as variáveis passam a apontar para o mesmo elemento, simultaneamente e, assim, as alterações feitas em uma variável que compartilha referência com outras refletem instantaneamente em todos os outros pontos do código que olhem para aquela mesma referência. Devem, portanto, ser trabalhados com bastante cuidado.

Ao mencionarmos a “localização” a que a referência aponta, não citamos exatamente onde o elemento se encontra, e isso foi proposital. No Solidity, ela deve ser declarada explicitamente. A isto chama-se *data location* ("localização do dado"), e ela pode ser:

- Em memória (*memory*) — utilizada quando o tempo de vida do elemento fica limitado ao período em que sua função está sendo utilizada, sendo encerrada logo ao término da execução. Este tipo de dado só pode ser declarado dentro do escopo de funções.
- Em armazenamento (*storage*) — utilizada quando o tempo de vida do elemento é limitado pelo tempo de vida do contrato, sendo mantido ativo enquanto o contrato estiver em execução. É uma *location* mais cara que o *memory*, quando consideramos as taxas em *Gas*, justamente por possuir um tempo maior de vida, o que também deve ser considerado em momento de desenvolvimento.
- *Calldata* — uma localização similar ao *memory*, porém, que garante que o dado referenciado não possa ser alterado.

São dois os principais tipos de dados de referência no Solidity: os vetores (*arrays*) e os *structs*, que compõem elementos a partir de tipos primitivos, permitindo a criação de tipos complexos.

1. *Arrays* — são conjuntos de dados de um mesmo tipo agrupados em uma única estrutura (chamada de vetor), em que cada um dos dados (chamado elemento) possui uma posição correspondente, única e fixa. Um elemento pode ser acessado em um *array* diretamente por sua posição (também chamada de

índice), considerando a primeira posição a de índice 0. *Arrays* podem ser criados possuindo tamanho fixo, definido no momento de sua declaração, ou podem ter tamanho dinâmico - o que é computacionalmente mais caro e, portanto, possui maiores taxas de *Gas* associadas. *Arrays* com referência na memória são sempre de tamanho fixo, e são criados pelo uso da palavra reservada *new*.

2. *Structs* — são estruturas complexas definidas pelo próprio usuário, criadas pela composição de diferentes tipos de dados, quando faz sentido agrupá-los em um mesmo elemento. É o caso, por exemplo, quando se deseja modelar logicamente uma entidade “aluno” — composto por informações como nome e número de matrícula, sendo o nome uma variável textual e o número de matrícula uma variável numérica. Cada uma dessas informações recebe o nome de “membro”. Linguagens de programação como o C possuem implementação de *structs* muito similar. Nela, no entanto, *structs* são considerados tipos de valor, e não de referência.

Mappings

Tipos de Mapeamento (*mappings*) são estruturas do tipo chave-valor, em que chaves podem ser de qualquer tipo único encontrado no Solidity (o que exclui *arrays*, *structs* e outros *mappings*), incluindo contratos, propriamente ditos, e os valores podendo ser de qualquer tipo, sem limitações. *Mappings* existem em uma diversidade de linguagens de programação; em algumas com esse mesmo nome e, em outras, chamados de “dicionários”. Eles funcionam como uma tabela de *hashes*, que retorna valores específicos para cada parâmetro de consulta. Apesar de não serem um tipo de referência eles podem ter suas localidades declaradas ou não, sendo estas obrigatoriamente do tipo *storage*.

6.4 Variáveis

De acordo com o *styleguide* do Solidity, nomes de variáveis devem seguir o padrão *mixedCase*, iniciada por uma letra minúscula, seguida por uma outra palavra com letra maiúscula, preferencialmente em inglês como em *amountPaid*, por exemplo. Variáveis representando *Enums* e *structs*, por sua vez, seguem o padrão *CapWords*, em que cada palavra é iniciada por uma letra maiúscula. Suas declarações são feitas pela enumeração do tipo da variável, seguido do seu nome.

Interessantemente, o conceito de variáveis nulas (*null*) ou indefinidas (*undefined*) não existe no Solidity, sendo que toda variável possui um valor padrão definido pela linguagem — e o tratamento de valores inesperados é feito por meio das chamadas "*revert functions*" (“funções de reversão”). Existe um operador no Solidity chamado *delete*, que inicializa a variável para seu valor padrão, o que é bastante útil.

O Solidity traz consigo, ainda, uma série de variáveis de âmbito "global", que podem ser acessadas a qualquer momento, dentro do código, sem a necessidade de instanciações ou qualquer tipo de parametrização prévia. Elas são usadas para a consulta de propriedades sobre a blockchain e seus componentes.

Os primeiros representantes deste grupo são as propriedades que dizem respeito aos blocos (*blocks*). São elas:

1. *block.basefee* — retorna a taxa-base o bloco atual. Este conceito foi incluído com a atualização *London*, e garante um valor mínimo de *Gas* requerido para a inclusão do bloco em questão, calculado dinamicamente, de acordo com o tamanho do bloco anterior;
2. *block.chainid* — retorna o *chain id* do bloco atual. Este é um parâmetro de segurança, que garante que um bloco seja processado uma única vez;

3. *block.coinbase* — retorna o endereço do responsável (pessoa ou entidade) responsável por minerar o bloco atual;
4. *block.difficulty* — retorna a dificuldade do bloco atual;
5. *block.gaslimit* — retorna o limite de *Gas* do bloco atual;
6. *block.number* — retorna o número do bloco atual;
7. *block.timestamp* — retorna o *timestamp* do bloco atual, em segundos, contados a partir de 1 de janeiro de 1970.

No contexto da mensagem, isto é, da chamada feita para o contrato via EVM, existem quatro propriedades importantes. Há de se prestar muita atenção ao fato de que contratos podem ser chamados de diferentes pontos, inclusive por outros contratos ou bibliotecas e que, portanto, estes valores podem ser alterados conforme o caso de uso. São eles:

1. *msg.data* — retorna os dados completos da mensagem;
2. *msg.sender* — retorna o endereço do remetente da mensagem;
3. *msg.sig* — retorna os quatro primeiros bytes da mensagem, servindo como um rápido identificador;
4. *msg.value* — retorna o valor (em Weis) sendo transferido na transação em questão.

O último contexto importante é o da transação. Aqui, duas são as principais propriedades utilizadas:

1. *tx.gasprice* — retorna o preço da transação devido às taxas cobradas como *Gas*;
2. *tx.origin* — retorna o endereço do responsável pelo envio da transação.

6.5 Tratamento de erros e exceções

O tratamento de erros e exceções no Solidity deve ser cuidadosamente planejado e implementado, visto o fato de as transações lidarem com divisas financeiras diretamente, além do

repetidamente citado caso da imutabilidade dos contratos. Da mesma forma, a linguagem possibilita diferentes formas de tratamento, através de funções específicas.

O primeiro modo de tratamento particular do Solidity para este tipo de situação inesperada a ser discutido são as funções de reversão — ou, simplesmente, **revert** — que têm a capacidade de retornar o estado do sistema ao estado imediatamente anterior à chamada da função, realizando a devolução de eventuais divisas já movimentadas, por exemplo, sem a necessidade de fazer esse tratamento via código.

O *revert* pode ser utilizado diretamente, retornando uma mensagem definida pelo usuário, como também pode ser utilizado em conjunto com um elemento de erro personalizado, definido pelo usuário (o que é uma opção mais barata em termos de *Gas*).

Um erro personalizado pode ser criado a partir da palavra reservada *error*, e sua definição. Uma vez criados, os erros personalizados podem ser usados sempre que a situação corresponder a eles — por exemplo, a criação de um erro de acesso não autorizado, disparado quando um usuário enviar credenciais incorretas para o contrato.

Classicamente, o *revert* é utilizado logo após uma verificação condicional, seguindo o padrão `if(!condição) revert(…)`. Como tal, ele não é utilizado para avaliar condições, sendo responsável apenas por, quando invocado, sinalizar a ocorrência de um erro e retornar o sistema ao estado anterior.

Um resultado similar para esse tipo de situação é alcançado através da diretiva **require**, esta sim capaz de realizar comparações. Sua própria sintaxe — `require(condição, “mensagem de erro”)` — permite indicar a condição necessária para que o código continue a ser executado, sendo capaz de disparar mensagens de erro, recebidas também como parâmetros.

Casos comuns de uso são a validação de entradas de usuário, validação de retornos de funções externas, validação de condições prévias à execução de trechos de código.

Internamente, o compilador do Solidity lança um erro do tipo `Error(string)`, tanto para o caso da ocorrência de uma falha com o *revert* como com o *require*.

Uma terceira maneira de se trabalhar com erros é através da diretiva **assert**. Ela é usada principalmente para tratar de invariantes e condições não esperadas durante o processamento, tais como a ocorrência de *underflow/overflow*, problemas com o gerenciamento de memória, mudanças de estado indesejadas ou mesmo como um gatilho de segurança para evitar catástrofes durante o processamento — como no caso de evitar alguma condição que jamais deveria ser possível.

Sua forma geral é `assert(condição)`, não sendo capaz de realizar o disparo de mensagens de erros, por exemplo. Erros identificados através da cláusula *assert* são tratados via um tipo de erro denominado `Panic(uint256)`.

Ambas as assinaturas de erros (*Error* e *Panic*) podem ser usadas dentro de diretivas *try/catch* que, por serem comuns à grande maioria das principais linguagens de programação em uso, não serão aqui discutidas. Blocos *catch* podem, portanto, ser trabalhados para tratarem essas exceções através das fórmulas `catch Error(string memory mensagem_de_erro)` e `catch Panic(uint codigo_do_erro)`.

6.6 Eventos

Um evento é definido, tecnicamente, como um membro de um contrato capaz de armazenar parâmetros, emitidos através de logs por blockchains. Normalmente são utilizados como sinalizadores de

mudança de estado na rede, e trechos de código que estejam subscritos a ele são capazes de disparar o processamento de algum tipo de lógica, quando a mudança ocorrer.

Eventos são criados a partir do uso da palavra reservada `event`, seguida de seu nome e seus parâmetros, caso existam — e estes representam os valores armazenados no *log*.

Uma vez criados, eles podem ser chamados em qualquer parte do código, através do uso da palavra reservada `emit` seguida do nome do evento desejado e da inclusão dos parâmetros necessários, na forma `emit nome_do_evento(parâmetros)`.

Durante a criação dos eventos, até três de seus parâmetros podem ser declarados utilizando o modificador `indexed`, e passam a fazer parte de uma estrutura especial dos logs chamada de “tópico” (*topic*), e não mais na porção reservada para guardar os dados do evento — usados, assim, como indexadores no *log* e facilitadores durante a busca por registros.

Eventos anônimos, isto é, sem tópicos, são mais baratos de serem armazenados, o que é um balanço que deve ser estudado durante o desenvolvimento dos contratos.

De acordo com o *styleguide* do Solidity, nomes de eventos devem seguir o padrão *CapWords*, em que cada palavra é iniciada por uma letra maiúscula, como em `TransactionPaid`, por exemplo.

6.7 Funções

O nome “função” pode representar tanto um *value type* no Solidity, como pode se referir à função, propriamente dita, que representa as possíveis ações a serem tomadas dentro do contrato, como é comum nas linguagens de programação.

Visto como um tipo por valor, o tipo *function* é típico das funções em geral, e são usados por variáveis que as armazenam, permitindo que sejam passadas como parâmetro ou retornadas como valor de retorno.

Funções, no Solidity, podem não retornar qualquer valor, como podem retornar um ou mais valores. Os tipos de retorno, quando existirem, devem ser explicitamente declarados, utilizando a palavra reservada `returns`, seguindo o padrão `function nome_da_funcao returns(tipo_de_retorno)`.

Funções vistas como tipos ou como implementações, propriamente ditas, podem ter suas visibilidades (*visibilities*) declaradas como internas (*internal*) ou externas (*external*), sendo que as primeiras só podem ser chamadas dentro do próprio contexto em que foram declaradas, o que inclui, além do próprio contexto, os contratos que herdam do contrato em que foram declaradas. Já as últimas podem ser chamadas por outros contextos, externos, como o nome sugere, por exemplo, por meio de transações.

Implementações podem ser públicas (tipo *public*) ou privadas (tipo *private*). Assim como em outras linguagens de programação, sua diferença se dá no contexto em que elas podem ser invocadas, sendo as implementações públicas aquelas visíveis a partir de qualquer lugar do código (podendo ser chamadas em qualquer lugar) e as privadas as que são acessíveis apenas dentro do contrato em que são especificadas.

Da mesma forma, implementações podem ser de três tipos: de visualização (declaradas pela palavra reservada `view`), puras (declarada pela palavra reservada `pure`), ou pagáveis (`payable`). Seus casos de uso são descritos a seguir:

1. Funções do tipo *view* são aquelas incapazes de modificar o estado do elemento a que se referem. Pelo contrário, elas são indicadoras de que o contrato sofreu uma alteração em seu estado. São usadas, por exemplo, para impedir que sejam

utilizados para a criação de novos contratos, para a realização de transferência de divisas financeiras entre contas, para a atribuição de valores a variáveis, trabalhar com chamadas de baixo nível, entre outros.

2. Funções do tipo *pure* são ainda mais restritas, e residem na expectativa de que são incapazes tanto de alterar o estado como mesmo de ter acesso a dados sobre ele. Nesse caso, além da lista já citada para as funções do tipo *view*, uma função *pure* é também incapaz de ler varáveis de estado e mesmo acessar variáveis globais como os membros `block`, `tx` e `msg`;

3. Funções *payable*, por sua vez, são aquelas para as quais subtende-se que determinada quantia de Ether será enviada durante sua invocação, capazes, portanto, de cobrarem divisas por seu uso. Este modificador é muito importante, visto que qualquer tentativa de envio de quantias financeiras para uma função sem esta marcação resultará em uma rejeição de sua execução.

Por fim, existem outros dois tipos de funções bastante específicos:

1. Funções de recebimento (*receive functions*) — representam um tipo especial de função, declarada sem a utilização da palavra reservada *function*, sem a declaração de parâmetros e sem qualquer tipo de retorno. Devem, obrigatoriamente, ser funções do tipo *payable*, e são invocadas por meio dos membros `.send()` ou `.transfer()`, usados, por exemplo, por variáveis de tipo *address*. Sua ausência pode impedir que o contrato receba dinheiro, mesmo que o usuário envie, explicitamente. Sua declaração é feita por meio do padrão `receive() external payable {}`.

2. Funções de *fallback* também são declaradas sem a utilização da palavra reservada *function* e, obrigatoriamente, devem ter suas visibilidades definidas como *external*. Funções de *fallback* são utilizadas quando, por algum motivo, uma função é

invocada externamente, porém, o nome e a assinatura não são reconhecidos pelo contrato como uma função válida ou, ainda, no caso de o contrato não possuir uma função de recebimento, como última alternativa (neste caso, ela obrigatoriamente deve receber o modificador *payable* para funcionar como recebedora de divisas). São declaradas por meio do padrão `fallback() external [payable]{}`.

De acordo com o *styleguide* do Solidity, na declaração de funções, primeiro são declarados os modificadores de visibilidade e, em seguida, os modificadores de mutabilidade. Seus nomes devem seguir o padrão *mixedCase* (iniciando com letra minúscula e, em seguida, por letra maiúscula) — por exemplo, `checkBalance` —, preferencialmente sendo nomeadas em inglês, sendo a mesma regra válida para a nomenclatura dos parâmetros.

Assim como ocorre com as variáveis, o Solidity possui funções de âmbito global, entre elas, a `blockhash(uint blocknumber)`, que recebe como parâmetro o número de um bloco qualquer, e retorna seu *hash*. Essa função só funciona para os últimos 256 blocos; caso contrário, retorna o valor zero.

Também uma função global, o `gasleft()` retorna a quantidade de *Gas* que resta disponível no contrato, representando, portanto, uma propriedade interna.

6.8 Building blocks do Solidity

A linguagem Solidity possui alguns artefatos que são verdadeiras peças-chave para seu desenvolvimento. São eles: o contrato, a declaração de interfaces, a criação de bibliotecas e a declaração de modificadores personalizados. A seguir, dedicaremos alguns parágrafos sobre cada um deles, em que apresentaremos suas finalidades e particularidades.

Contratos

Uma declaração de contrato (*contract*), no Solidity, possui o mesmo efeito da declaração de uma classe (*class*) em uma linguagem orientada a objetos, ainda que seus conceitos sejam diferentes. É graças a ele que o código é, efetivamente, implementado. Ele é composto por todos os elementos vistos neste capítulo, tais como funções, modificadores, erros, e também pelas chamadas variáveis de estado (*state variables*), que são variáveis declaradas no escopo do contrato — e, portanto, com localização no *storage*, já que são válidas enquanto o contrato estiver em execução.

Um contrato é criado via seu construtor, uma declaração especial indicada pela palavra reservada `constructor`, podendo ou não ter parâmetros, executada uma única vez — no momento de sua criação, quando o contrato passa a existir durante a execução. Sua declaração segue o padrão `constructor(parâmetros)`, sem a necessidade de declarar seu nome. Assim como em outras linguagens orientadas a objeto, a declaração do construtor não é obrigatória quando ele não possui parâmetros ou tratamentos especiais.

De acordo com o *styleguide* do Solidity, a ordem de declarações dentro de um contrato deve ser a seguinte:

1. Declarações de tipos especiais, tais como *erros* e *structs*;
2. Declarações de variáveis de estado;
3. Declarações de eventos;
4. Declarações de funções, agrupadas na ordem: construtor, função de recebimento (se existir), função de *fallback* (se existir), funções externas, funções públicas, funções internas e funções privadas. Funções do tipo *view* e *pure*, dentro de cada grupo, devem ser as últimas a serem declaradas.

Contratos com Solidity podem trabalhar com o conceito de **herança**, tema este recorrente e bem conhecido da Orientação a Objetos. Como veremos, seus principais conceitos são também parecidos. A herança é sinalizada pelo uso da palavra reservada `is`, seguindo a fórmula `contract nome_do_contrato is contrato_pai(parâmetros){}`.

Um *override* de função existe quando um contrato sobrescreve a implementação de um método definido no contrato-pai. Para que isso possa ocorrer, a função do contrato-pai deve, obrigatoriamente, ser declarada utilizando o modificador *virtual*, enquanto a função do contrato-filho, por sua vez (a que está realizando o *overriding*), deve também utilizar uma palavra reservada — neste caso, `override`. Por serem conceitos bastante comuns e populares, com os quais programadores de todos os níveis já foram expostos em algum momento, não entraremos aqui em detalhes, além de mencionar esta possibilidade.

Para o caso de contratos que trabalhem com herança, caso o construtor do contrato-pai requeira parâmetros, estes devem ser enviados diretamente na chamada da função ou podem ser declarados explicitamente durante a implementação de seu próprio construtor. Nesse caso, o contrato-base deve ser invocado.

Diferentemente do Java, por exemplo, que talvez seja a principal das linguagens de programação orientadas a objetos mais populares, o Solidity trabalha com o conceito de herança múltipla e, neste caso, trabalha com a chamada `Linearização C3`.

O conceito da `Linearização C3` existe há mais de duas décadas, e tem por objetivo definir a precedência das funções/métodos em casos de herança múltipla através de um algoritmo especializado. Falamos aqui que a ordem de precedência dos contratos-pai, no Solidity, é definida pela ordem em que foram declarados a partir da palavra `is` — que deve ir do contrato mais básico até o contrato com maior número de heranças. Por esta abordagem, os contratos mais à direita realizam um *override* nos contratos mais à esquerda, prevalecendo, portanto. Apesar de ser um conceito simples de se

entender, ele pode se tornar confuso em implementações mais complexas, o que se torna um importante ponto de atenção.

Até aqui neste livro, repetimos diversas vezes que contratos são imutáveis. Apesar disso, eles podem ser destruídos, passando a ser inválidos e, portanto, enviando seu eventual saldo financeiro para outra localidade, através do método `selfdestruct(address)`, que deve ser utilizado com extrema cautela, visto invalidá-lo completamente. Outro cuidado a se tomar é o de que um contrato que em algum momento no futuro precise ser destruído deve ter implementada uma função que execute o `selfdestruct` previamente, o que também é um ponto de atenção durante a fase de desenvolvimento.

Um tipo especial de contrato é o contrato abstrato, definido pela palavra reservada `abstract`. Tal como uma classe abstrata, ele não pode ser instanciado, mas pode ser herdado e suas funções podem ser ou não implementadas pelos contratos filhos. Sua declaração segue o padrão de nomenclatura `abstract contract nome_do_contrato{}`. Sua função principal é a de "exprimir um conceito", indicando os dados e ações trabalhados por um contrato genérico, porém, para serem utilizados, precisam ser herdados por contratos derivados, especializados. Seu conceito é bastante conhecido da Orientação a Objetos, e suas aplicações facilitam, por exemplo, tarefas como o reúso de código e separação de contextos em um nível lógico.

Interfaces

Interfaces são artefatos similares ao contrato abstrato, e também são bastante conhecidas por programadores. Tal como uma base abstrata, uma interface não pode ser instanciada, porém, difere-se do contrato abstrato por não implementar funções explicitamente (limitando-se a declará-las) ou conter variáveis de estado. Elas são declaradas no Solidity a partir do padrão `interface nome_da_interface{}` e contratos podem herdar de interfaces da mesma forma que herdam de outros contratos.

Interfaces são utilizadas como "regras" de funcionamento entre diferentes contextos em uma aplicação e mesmo entre subsistemas; são ferramentas poderosas utilizadas na diminuição do acoplamento entre os diferentes artefatos e, por isso, são extremamente populares na implementação de padrões de desenho e adoção de boas práticas para a escrita de código.

Bibliotecas

Bibliotecas possuem a mesma finalidade de um contrato-pai de disponibilizar funções que possam ser utilizadas por contratos que herdem dele. A principal diferença é a de que uma biblioteca é publicada uma única vez, em uma localidade definida, e seu código é simplesmente reutilizado pela EVM.

Além disto, uma biblioteca não pode possuir variáveis de estado, não pode ser herdada nem herdar de outros contratos, não é capaz de receber pagamentos em Ether e também não pode ser destruída. Sua implementação é feita pela declaração da palavra reservada `library`, seguida do nome da biblioteca, seguindo o padrão `library nome_da_biblioteca{}`.

A nomenclatura de contratos e bibliotecas deve seguir o padrão *CapWords*, em que todas as palavras que compõem seu nome iniciam por uma letra maiúscula. O *styleguide* do Solidity não dá direcionamentos sobre interfaces, porém, recomendamos que sejam tomadas as mesmas práticas.

Modificadores personalizados

A linguagem Solidity permite a criação de alguns tipos de modificadores personalizados, que podem ser utilizados livremente por funções para garantirem o cumprimento de determinadas premissas antes de que a função seja executada. O exemplo clássico é o da utilização de um modificador `isOwner`, cujo uso garante que apenas o dono do contrato seja capaz de executar determinada função. Eles são declarados pela palavra reservada

`modifier`, seguida de seu nome, no padrão `modifier nome_do_modificador {}`. Sua implementação deve ser capaz de verificar uma determinada condição é atendida e, caso contrário, lançar uma exceção. Aqui, obrigatoriamente existe a declaração de uma diretiva `_;`, que indica ausência de ação, permitindo que o restante do código da função seja executado, ou um retorno explícito, implementado pela diretiva `return`.

6.9 Documentação do código

O Solidity apresenta, nativamente, um modo de documentação de seu código baseado em comentários colocados anteriormente à declaração de funções e variáveis de retorno. O formato é denominado *NatSpec*, nome que provém da expressão *Ethereum Natural Language Specification Format* ("Formato de Especificação em Linguagem Natural do Ethereum").

De acordo com a documentação oficial da linguagem, o *NatSpec* foi influenciado pelo *Doxygen*, utilizado, por exemplo, na documentação de código escrito em Java. O código incluído como comentário de acordo com o padrão *NatSpec* é compilado em conjunto com o restante do contrato e pode ser parseado para leitura, tanto para o usuário como para o desenvolvedor, já que são geradas documentações focadas em ambos os públicos.

De forma geral, a documentação é realizada por meio de comentários iniciando por três barras (`///`) - utilizados para comentarem uma única linha — ou iniciados por uma barra e dois asteriscos (`/**`) e finalizados por um asterisco e uma barra invertida (`*/`), que podem ser incluídos acima de elementos tais como contratos, interfaces, funções ou eventos, e que são responsáveis por conterem as informações sobre o código. Seus detalhes são incluídos por meio das seguintes *tags*:

1. `@title` — informa o título, capaz de descrever o elemento;
2. `@author` — descreve o autor responsável por desenvolver aquele trecho de código;
3. `@notice` — explica o funcionamento daquele trecho de código;
4. `@dev` — dá detalhes técnicos do desenvolvimento, em documentação disponibilizada para desenvolvedores;
5. `@param` — documenta os parâmetros utilizados;
6. `@return` — descreve as variáveis de retorno daquele trecho de código;
7. `@inheritdoc` — possibilita trabalhar com o conceito de herança, na documentação, herdando do documento-pai todas as *tags* que, eventualmente, não foram preenchidas.

Uma vez preenchidos os dados correspondentes, o próprio compilador *solc* pode ser utilizado para gerar a documentação, através do comando a seguir, que considera o nome do arquivo do contrato como `TestContract.sol`.

```
solc --userdoc --devdoc TestContract.sol
```

Conclusão

Neste capítulo vimos os principais componentes do Solidity e começamos, finalmente, a ter uma ideia de seu funcionamento em nível arquitetural como também de sintaxe. Os temas citados aqui são extremamente importantes e com certeza serão revisitados pela leitora e pelo leitor durante suas experiências durante a escrita de seus *smart contracts*, por influírem, diretamente, em questões de segurança e consumo de *Gas* dos contratos — apenas a título de exemplo.

Assim como outras linguagens de programação, o Solidity também possui uma coleção de padrões de implementação, que mostraram bom funcionamento em aplicações práticas, e não é difícil encontrá-los em livros e sites especializados.

Na bibliografia deste livro, deixamos o endereço do repositório Git de Franz Volland, que reuniu em um trabalho bastante primoroso e muito bem documentado alguns padrões utilizados, dividindo-os em padrões de comportamento, padrões de segurança, padrões econômicos e padrões com vista a permitirem a atualização de protocolo com mais facilidade, com acessos ao código e detalhes sobre a implementação.

Apesar de o próprio repositório conter uma informação de que ele está desatualizado (visto ter sido construído para a versão 0.4.20 do Solidity), sugerimos uma visita aos exemplos ali citados, que certamente contribuirão, de alguma forma, com o desenvolvimento de contratos mais atualizados.

No próximo capítulo vamos finalmente implementar alguns *smart contracts*, e colocaremos em prática a teoria vista aqui. Ficará clara a importância da ordem e da correção das declarações, e preocupações como a localização — incomuns em linguagens de programação populares —, por exemplo, passarão a fazer sentido. Após a prática, decisões como essa se tornarão automáticas para o desenvolvedor de contratos inteligentes que apenas começa a surgir!

Capítulo 7
Implementando Smart Contracts e Tokens

7.1 Introdução

Tendo sido formalmente apresentados à linguagem de programação Solidity no capítulo anterior, agora podemos, finalmente, começar a implementar nossos contratos inteligentes, simulando seu funcionamento dentro de uma blockchain real.

Este capítulo foi dividido na forma de exemplos, com diferentes graus de dificuldade que buscam, gradualmente, aplicar as funcionalidades disponíveis na linguagem, demonstrando seu potencial de uso na criação de *smart contracts*.

Não é demais lembrar que este não é um curso de programação e que, portanto, não nos aprofundaremos nas explicações sobre a lógica de programação aplicada ou padrões utilizados em orientação a objetos.

Iniciaremos nossa jornada definindo os padrões de organização de código e declarações utilizados pela comunidade do Solidity ao redor do globo. Ainda que sua aplicação não seja obrigatória, é uma prática muito positiva e deve ser encorajada, de forma a obtermos um padrão uniforme de escrita de código, o que torna tanto sua manutenção como seu compartilhamento muito facilitados, visto permitir com que todos os desenvolvedores compartilhem os mesmos preceitos.

Em seguida faremos a implementação de dois *smart contracts*, com diferentes graus de dificuldade, e seguiremos com a implementação de seus testes unitários — que, quando executados, permitem não apenas garantir seu correto funcionamento, mas apresentam uma prévia de seu funcionamento em uma blockchain real através do Ganache.

Nestas seções veremos na prática a aplicação dos artefatos apresentados no capítulo anterior, a importância das declarações de tipos e localizações de variáveis e, também, a aplicação de diretivas `require` e `assert`, críticas para a segurança de nossos contratos.

Mais adiante neste capítulo exemplificaremos a implementação de dois *tokens*, nos padrões ERC-20 e ERC-721, que são extremamente úteis para o desenvolvimento de projetos no Ethereum. Como já vimos, *tokens* ERC-20 são usados como "moeda de troca" pelo consumo de um serviço na blockchain (normalmente publicado na forma de um *smart contract*), enquanto que *tokens* ERC-721 representam a "posse" sobre determinado ativo digital.

Finalmente, encerramos esta parte apresentando considerações sobre a otimização dos custos de *Gas* durante a implementação de contratos, discussão esta extremamente relevante para qualquer profissional que deseje se aventurar pelo desenvolvimento de *smart contracts* para a Web 3.0.

7.2 Guia de estilo e organização do código

O *styleguide* do Solidity apresenta direcionamentos importantes para garantir a consistência da implementação de seus códigos. Como boa prática, seguiremos suas regras no desenvolvimento de nossos contratos. A seguir, damos uma lista dos principais direcionadores de estilo a serem considerados:

1. A indentação do código deve ser feita utilizando quatro espaços simples, devendo evitar-se utilizar o `Tab` para tal;

2. Implementações de diferentes elementos primários (contratos e bibliotecas, por exemplo) devem ser separadas por duas linhas em branco;

3. Dentro de um contrato, as implementações de elementos internos devem ser separadas por uma linha em branco;
4. Uma linha deve conter no máximo 79 caracteres. Caso seja necessário, uma nova linha deve ser inserida, respeitando o espaçamento de indentação de quatro espaços simples;
5. Parâmetros e argumentos, quando separados em linhas diferentes, devem ser posicionados na forma de um elemento por linha;
6. Na declaração de contratos, funções e outros elementos que utilizem separação por chaves ({}), a chave de abertura deve ficar posicionada na primeira linha, e a chave de fechamento deve estar isolada na última linha.

Da mesma forma, também organizaremos nosso contrato de acordo com as diretivas dadas pelo guia de estilo, que prevê a seguinte ordem para os elementos:

1. Declaração sobre a licença de uso do contrato — utilizados aqui os padrões do *SPDX*, que podem ser consultados diretamente em sua documentação oficial, citada nas referências, no capítulo final;
2. Declaração da diretiva *Pragma* — define quais versões do compilador podem ser utilizadas para compilar o código apresentado; ela pode incluir uma versão específica do compilador, determinar as versões mínimas, ou estabelecer uma faixa de versões de trabalho. Esta declaração é extremamente importante, visto que o Solidity é uma linguagem em constante evolução, e que novas versões são disponibilizadas constantemente para o público;
3. Declarações do tipo *import*;
4. Declarações de *interfaces*;
5. Declarações de bibliotecas;

6. Declarações dos contratos.

7.3 Smart Contract - Cadastro de alunos

A seguir, implementaremos nosso primeiro contrato, dedicado a realizar o controle da matrícula de alunos na disciplina “Programação para Blockchain”. Para isso, dentro da pasta *contracts*, criada no capítulo 5, criamos um arquivo, chamado `StudentsContract.sol`.

Inicialmente, como mostrado a seguir, declaramos o tipo de licença do nosso contrato (usaremos aqui o padrão MIT), declaramos a diretiva *pragma* para a versão do compilador — considerada aqui a nossa versão — 0.8.12 — e, finalmente, declaramos nosso contrato, com o nome `StudentsContract`.

```
// SPDX-License-Identifier: MIT
pragma solidity = 0.8.12;

contract StudentsContract{

}
```

Primeiramente, definimos os dados que identificam cada um de nossos alunos; para isso, dentro do contrato, declaramos um *struct* de nome `Student`, em que declaramos como membros uma variável do tipo String — chamada `studentName` — que armazena o nome do aluno, e outra do tipo *uint* — chamada `studentAge` —, que armazena sua idade.

```
struct Student{
    string studentName;
    uint studentAge;
}
```

Considerando que um aluno deveria ser capaz de ser inscrito uma única vez no curso, devemos ser capazes de controlar, de alguma maneira, quais alunos já estão matriculados. Para isso, utilizaremos um *mapping*, que mapeará os usuários matriculados a partir dos endereços que dispararam a chamada aos contratos. Para cada endereço passado como chave, ele retorna o aluno como valor, e o chamaremos de `enrolledStudents`. Por ser um elemento de uso exclusivo dentro do contrato, podemos definir sua visibilidade como privada.

```
mapping(address=>Student) private _enrolledStudents;
```

A primeira função que vamos desenvolver é, justamente, aquela que busca o registro correspondente ao endereço da requisição no *mapping* `enrolledStudents`. Declararemos uma função chamada `getEnrolledStudentByAddress`, que recebe como parâmetro uma variável do tipo *address*.

Deixaremos a visibilidade desta função como pública, porque queremos que ela seja acessada externamente. Quanto à mutabilidade, a classificaremos como uma *view* — isso porque ela deve ser capaz de ler valores de variáveis (o que é impossível para uma *pure*), porém, não deve ser capaz de alterá-los.

Como retorno, ela devolve uma variável do tipo `Student` que, por se tratar de um *struct* e, portanto, um tipo de dado de referência, deve ter sua localização declarada explicitamente. Como o dado não será acessado após a conclusão da execução da função, o deixaremos localizado em memória. Assim, sua declaração é:

```
function getEnrolledStudentByAddress(
            address _studentAddress
        )

    public
    view
        returns (Student memory){
```

```
}
```

Sua implementação é bastante simples, bastando retornar o valor correspondente ao endereço contido no *mapping*. Seguindo o conceito de chave-valor, isso pode ser conseguido através do acesso via colchetes, indicando aí o endereço (a chave) desejada:

```
return _enrolledStudents[_studentAddress];
```

A declaração completa da função é, portanto:

```
function getEnrolledStudentByAddress(address _studentAddress)
    public
    view
    returns (Student memory){
        return _enrolledStudents[_studentAddress];
}
```

Lembrando-nos de que no Solidity não existe o conceito de variáveis com valores nulos ou indefinidos, caso um endereço não cadastrado em nosso *mapping* seja passado como parâmetro, o retorno será, de fato, um elemento do tipo `Student`, porém preenchido com os valores-padrão de cada tipo de dado que o compõe; isto é: um texto vazio, no caso do nome, e o valor zero, no caso da idade. Um elemento `Student` com essas duas condições (nome vazio e idade zero) é entendido, portanto, como um membro inexistente no *mapping*.

Com base nisso, podemos criar duas funções distintas: uma que verificará se a idade é maior do que zero, e outra, que verificará se o nome não está vazio.

A primeira das funções, que receberá o nome `isStudentAgeValid`, recebe como parâmetro um valor inteiro maior do que zero — portanto, um inteiro sem sinal. Por representarmos idades, um inteiro positivo é suficiente, portanto, o parâmetro será do tipo *uint* que, por ser um tipo de dado de valor, não necessita de demais declarações.

A função será usada exclusivamente no escopo do contrato, portanto, privada e também não fará algum tipo de consulta ou manipulação de variáveis, podendo ser declarada como pura. Seu retorno é um valor booleano indicando se o valor é ou não maior do que zero. Sua implementação é a seguinte:

```
function isStudentAgeValid(uint _studentAge)
    private
    pure
    returns(bool){
        return _studentAge > 0;
}
```

Da mesma forma, podemos implementar a função `isStudentNameValid`, que receberá uma String como parâmetro — este, sim, um tipo de dado de referência, cuja localização precisa ser declarada.

Por ser um dado que não existirá fora do contexto desta função, podemos declará-lo como localizado na memória. Os demais modificadores seguem a mesma linha de raciocínio da função `isStudentAgeValid` e, portanto, uma função privada, pura, que retorna um valor booleano.

Se, na função anterior, a comparação para a idade do aluno foi fácil de ser implementada, visto que precisamos fazer uma comparação simples entre dois números, utilizando o operador maior que (>), a verificação do nome é um pouco mais complexa. Isto, porque o Solidity não possui implementações próprias para trabalhar com Strings (apesar de que isso possa ser feito facilmente pela utilização de bibliotecas).

Uma String é, por definição, um vetor de bytes, e pode, portanto, ser convertida explicitamente para esse tipo. Uma vez que vetores são tipos de dado de referência, precisamos explicitar também sua localização — e aqui utilizaremos a memória, mais uma vez, levando em conta que não o consumiremos mais após o uso. A

atribuição do valor à nova variável, chamada `tempString`, pode ser feita de acordo com a seguinte declaração:

```
bytes memory tempString = bytes(_studentName);
```

Finalmente, podemos incluir a cláusula de retorno. Caso a String original seja vazia, isso é refletido no vetor como um vetor vazio — isto é, de tamanho zero. A implementação completa da função é:

```
function isStudentNameValid(string memory _studentName)
    private
    pure
    returns(bool){
        bytes memory tempString = bytes(_studentName);
        return tempString.length>0;
}
```

Nossa quarta função, chamada `isStudentValid`, terá como objetivo definir se o um aluno é ou não válido, com base em seu nome e sua idade — aqui, consideramos válido um aluno que tenha um nome válido e que tenha uma idade válida. Como acabamos de declarar outras duas funções que façam estas comparações, individualmente, podemos agora reaproveitá-las.

Nossa função recebe como parâmetro um elemento `Student` que, por ser do tipo *struct* é um tipo de referência, cuja localização deve ser informada — e aqui mais uma vez trabalharemos com a memória visto não necessitarmos utilizar seu valor fora deste contexto. Como nos outros casos, é também uma função privada, pura, que retorna um valor booleano.

Seu retorno é uma operação `AND` das funções `isStudentNameValid` e `isStudentAgeValid`, conforme implementação mostrada a seguir:

```
function isStudentValid(Student memory _student)
    private
    pure
    returns(bool){
        return isStudentNameValid(_student.studentName) &&
```

```
            isStudentAgeValid(_student.studentAge);
    }
```

Nossa última função e, também, a mais importante, é aquela responsável por realizar o cadastro do aluno na base de alunos matriculados. Ela deve ser capaz de receber como parâmetros uma String indicando o nome do aluno — chamada aqui de `studentName` — e um número inteiro (sem sinal), com sua idade — `studentAge`.

Novamente partimos do princípio de que uma String é um tipo de dado de referência e, portanto, que exige a declaração de localização — e aqui, mais uma vez, o deixaremos localizado na memória, visto seu uso estar restrito ao contexto em que se encontra. Por ser uma função que será acessada externamente, alteramos seu modificador de visibilidade para público.

Sua declaração é a seguinte:

```
function enrollStudent(
    string memory _studentName,
    uint _studentAge
)
    public {

}
```

Para sua implementação, primeiramente precisamos garantir que o nome e a idade são dados válidos. Isso é conseguido através de diretivas `require`, que permitem, ainda, a inclusão de um texto com detalhes sobre o erro.

Para a inclusão destas validações, vamos nos aproveitar das funções criadas anteriormente, que verificam, individualmente, tanto a validade do valor informado para o nome, como o valor informado para a idade, conforme implementação a seguir:

```
require(
    isStudentNameValid(_studentName),
    "Name must be informed"
);
```

```
require(
    isStudentAgeValid(_studentAge),
    "Age must be informed"
);
```

Considerando que ambos os parâmetros tenham sido aprovados para uso, precisamos tomar o cuidado, também, de não permitir a inclusão de um endereço já existente (isto é, alunos repetidos). Aqui, faremos uma composição de funções.

Primeiro realizamos uma busca em nosso *mapping* pelo aluno correspondente ao endereço que iniciou a execução do contrato — e usamos aqui a variável global `msg` para descobrir o remetente. Em seguida, precisamos garantir que o aluno que retornou da nossa consulta ao *mapping* não seja válido — ou seja, não possui os valores-padrão de cada tipo de dado.

Isso é conseguido através da seguinte implementação:

```
require(
    !isStudentValid(
        getEnrolledStudentByAddress(msg.sender)
    ),
    "Student already enrolled"
);
```

Caso as três condições tenham sido válidas, podemos prosseguir a execução do código.

Inicializamos uma variável do tipo `Student`, localizada na memória, já que não precisa ser válida fora do contexto desta função. Em seguida, passamos para ela, como valores de nome e idade os parâmetros recebidos `studentName` e `studentAge`, conforme a seguinte implementação:

```
Student memory student;
student.studentName = _studentName;
student.studentAge = _studentAge;
```

Na sequência e, como boa prática de segurança, podemos checar se a variável `student` é válida, antes de ser incluída a nosso *mapping*. Aqui, usamos a diretiva `assert`, e a função `isStudentValid`, declarada anteriormente:

```
    assert(
        isStudentValid(student)
    );
```

Estando tudo correto, podemos incluir o endereço do solicitante em nosso *mapping*, a partir da variável global `msg`. Aqui, associamos o valor `student` (um *struct*) ao *mapping* `enrolledStudents` através da chave representada pelo endereço do remetente da mensagem, isto é, o endereço que solicitou a execução do contrato — o remetente.

```
    _enrolledStudents[msg.sender] = student;
```

A implementação completa de nossa função é:

```
  function enrollStudent(
        string memory _studentName,
        uint _studentAge
    )

        public {

            require(
                isStudentNameValid(_studentName),
                "Name must be informed"
            );

            require(
                isStudentAgeValid(_studentAge),
                "Age must be informed"
            );

            require(
                !isStudentValid(
                    getEnrolledStudentByAddress(msg.sender)
                ),
                "Student already enrolled"
```

```
        );

        Student memory student;
        student.studentName = _studentName;
        student.studentAge = _studentAge;

        assert(
            isStudentValid(student)
        );

        _enrolledStudents[msg.sender] = student;
    }
```

7.4 Testando nosso Smart Contract

Conforme vimos no capítulo 4, o efeito imediato da compilação de um *smart contract* é a geração de seu *bytecode*, este, interpretado pela EVM para ser executado. A partir da compilação, somos capazes também de gerar uma estrutura chamada ABI — que também já foi mencionada — que representa, em um formato JSON, a estrutura de entradas e saídas de nosso contrato e que permite sua utilização por entidades externas, como o web3.js.

Nesta seção, executaremos nosso contrato utilizando o servidor *Ganache*, que foi instalado no passo a passo do capítulo 5, como um simulador de nossa blockchain, que atuará como a cadeia de blocos sobre a qual publicaremos nosso contrato. Para tanto, basta que o executemos e deixemos funcionando no modo "Ethereum".

Como primeiro passo, vamos inicialmente alterar nosso arquivo `compile.js` , criado no capítulo 5, para que ele compile nosso contrato e seja capaz de exportar tanto o ABI como seu *bytecode* correspondente.

Neste ponto, podemos remover as requisições feitas para a biblioteca *Chalk*, uma vez que não trabalharemos com a impressão

de textos no console.

As declarações de import que restarão são:

```
const path = require('path');
const fs = require('fs');
const solc = require('solc');
```

Em seguida, declaramos o nome do nosso contrato — `StudentsContract.sol` — como uma constante, cujo valor será reutilizado ao longo do código e que facilitará a manutenção de nosso arquivo no futuro. Seguimos declarando o caminho até ele e fazemos sua leitura a partir do `fs` do Node.js.

```
const contractName = 'StudentsContract.sol';

const contractPath = path.resolve(__dirname,
                                  '../contracts',
                                  contractName);

const source = fs.readFileSync(contractPath, 'utf8');
```

O próximo passo é preparar o JSON de compilação, seguindo o padrão especificado na documentação do compilador. Vamos chamá-lo de `compilerInput`.

Este é um parâmetro extremamente importante, que define, além da linguagem e da indicação do contrato, parâmetros que dizem respeito ao processo de compilação em si, como parâmetros ligados à utilização do otimizador de código. Aqui, utilizaremos a declaração mais genérica possível, sem incluir opções ou tratamentos especiais.

É importante notar que este formato pode mudar de acordo com a versão do compilador. Aqui apresentamos a versão válida para a versão 0.8.12 do *solc*, que segue exatamente o que está disposto na documentação oficial. Caso você esteja com uma versão diferente, é possível que as declarações sejam alteradas — e, para tanto, indicamos a consulta à documentação oficial do Solidity.

```
const compilerInput = {
    language: 'Solidity',
    sources:
    {
        'StudentsContract.sol':
        {
            content: source
        }
    },
    settings:
    {
        optimizer:
        {
            enabled: true
        },
        outputSelection:
        {
            '*':{
                '*':['*']
            }
        },
    }
};
```

Como próximo passo, fazemos a compilação do nosso contrato, utilizando o comando `compile`, do *solc*. Enviamos como parâmetro o nosso JSON declarado anteriormente e recebemos como resposta um outro JSON com o retorno do processo. Dentro dele, é possível extrair o contrato no modo com que é utilizado pela EVM.

```
const inputString = JSON.stringify(compilerInput);

const output = JSON.parse(
                    solc.compile(inputString)
                );

const contractEVM = output
                .contracts[contractName]
                ['StudentsContract'];
```

Por fim somos capazes de acessar diretamente a ABI e o *bytecode* de nosso contrato, diretamente do nosso contrato.

Aqui, exportaremos ambos os valores, para que possamos utilizá-los em outro arquivo.

```
const abiJSON = contractEVM['abi'];

const bytecode = contractEVM
                    ['evm']
                    ['bytecode']
                    ['object'];

module.exports = {abiJSON,bytecode};
```

No *VS Code*, criamos uma pasta chamada `test` dentro da raiz de nossa estrutura e, dentro dela, criamos o arquivo `StudentsContractTest.js`.

Inicialmente declaramos a requisição para o uso das bibliotecas `assert` — usada pelo *Mocha* para a realização dos testes unitários — e web3 — esta, utilizada como interface entre nosso contrato e a linguagem de programação JavaScript.

A requisição do web3 é feita para uma constante Web3, com letra maiúscula, visto que a instanciaremos mais à frente. Também fazemos a requisição do nosso ABI e do *bytecode*, exportados anteriormente do arquivo `compile.js`.

```
const assert = require('assert');

const Web3 = require('web3');

const {abiJSON,bytecode} =
    require('../scripts/compile.js');
```

Finalmente, utilizaremos o *Ganache* para simular o funcionamento de uma blockchain. Ao abrirmos o executável no modo "Ethereum", iniciamos uma blockchain fictícia em nossa máquina local. Seu endereço fica disponibilizado na tela principal da aplicação, sob o

nome *RPC Server*, normalmente utilizando a porta 7545. Este valor será utilizado para declararmos nosso provedor e o utilizarmos na instanciação do web3.

```
const provider =
   new Web3
   .providers
   .HttpProvider('HTTP://127.0.0.1:7545');

const web3 = new Web3(provider);
```

Aqui temos nossa blockchain devidamente preparada para uso, através do web3.

Como já vimos, o *Ganache* nos proporciona uma série de endereços fictícios que simulam contas do Ethereum, cada uma com uma quantidade inicial de Ether determinada — na versão atual, por padrão, uma quantidade igual a 100 ETH.

Nós usaremos estas contas (dadas na forma de endereços) tanto para publicarmos o contrato na blockchain como para interagirmos com ele. Assim, simplesmente declaramos quatro variáveis que serão utilizadas durante nossos testes, referentes à lista de endereços disponíveis, alguns endereços em específico, e o contrato publicado.

```
let accounts;
let account_1;
let account_2;
let studentsContract;
```

Utilizaremos agora uma notação do próprio *Mocha* que permite a realização de ações antes da execução de cada teste. Essas declarações são enviadas dentro de um bloco de função chamado `beforeEach`.

Antes da execução de cada teste, executaremos os seguintes passos, em ordem:

1. Consultaremos a lista de endereços disponíveis no *Ganache*;

2. Selecionaremos os dois primeiros endereços (e aqui a escolha é aleatória). Um deles será associado à variável `account_1` e o outro, à `account_2` ;

3. Faremos a publicação do nosso contrato a partir da ABI e do *bytecode* recuperados no arquivo `compile.js` . A publicação será feita a partir do endereço da variável `account_1` , e enviaremos, por padrão, uma quantidade de *Gas* equivalente a 1 milhão de weis.

A implementação desses passos é dada a seguir:

```
beforeEach(async()=>{
    accounts = await web3.eth.getAccounts();
    account_1 = accounts[0];
    account_2 = accounts[1];
    studentsContract =
        await new web3.eth.Contract(abiJSON)
        .deploy({data: "0x"+bytecode})
        .send({from: account_1, gas: 1000000});
});
```

Por fim, seguimos com a escrita dos nossos testes — declarados seguindo os padrões do próprio *Mocha* — conforme descrito nos próximos parágrafos.

Dentro do contexto "*Students Contract*", escrevemos nosso teste "*Should return a valid student*", que vai fazer a inclusão de um novo aluno através da função `enrollStudent` , definida em nosso contrato, que recebe como parâmetros o nome e a idade do aluno, e que será enviada a partir do primeiro endereço disponível (armazenado na variável `account_1`).

Como em toda transação Ethereum, precisamos enviar um valor de *Gas* correspondente, que será utilizado para a inclusão da transação no bloco.

Em seguida, consultaremos nosso elemento *mapping* definido no contrato, para verificar se o aluno foi realmente incluído aí,

realizando a consulta através do valor do primeiro endereço, utilizando a função `getEnrolledStudentByAddress`, também definida em nosso contrato.

Se tudo deu certo, ao retornarmos o valor do aluno passado pelo primeiro endereço, podemos comparar se o nome e a idade, retornados, estão de acordo com o esperado.

Para interagirmos com as funções do contrato, utilizaremos a biblioteca web3, através da lista de métodos disponíveis em sua propriedade `methods`, como segue.

```
describe('Students Contract', async() => {
    it('Should return a valid student', async () =>{

        let name = 'Joao Kuntz';
        let age = 36;

        await studentsContract.methods
            .enrollStudent(name, age)
            .send({from: account_1, gas: 1000000})

        let student = await studentsContract.methods
            .getEnrolledStudentByAddress(account_1)
            .call();

        assert.strictEqual(student[0], name);
        assert.strictEqual(student[1], age.toString());
    });
});
```

Para executar este teste, pela linha de comando e, estando na raiz de nosso projeto, executamos o comando a seguir, que fará a publicação do contrato, definida na diretiva `beforeEach` e, em seguida, executará o cenário.

```
npx mocha
```

A mensagem retornada pelo console, espera-se, mostrará que o teste foi aprovado, indicando também seu tempo de execução, em

milissegundos.

```
Should return a valid student (666ms)
```

Se observarmos o *Ganache*, veremos que a primeira conta (utilizada para a realização da publicação do contrato e também para a inclusão de um aluno novo) já não apresenta mais o saldo inicial de 100 ETH, tendo em vista as taxas pagas em forma de *Gas* para publicação e consumo.

Olhando em seguida para a aba *Blocks*, vemos que ao menos um bloco foi adicionado à nossa blockchain, e aí temos indicações sobre o *hash* do bloco, os endereços de interesse, quantidade de *Gas* usado, entre outros. Na aba *Transactions*, vemos detalhes das nossas transações incluídas no bloco (criação e consumo dos nossos contratos).

De forma didática, podemos escrever um segundo teste para verificar o que ocorre quando tentamos consultar um estudante a partir de um endereço que não foi utilizado para cadastrar alunos. Esperamos que sejam retornados os valores padrão — uma String vazia para o nome, e o valor zero para a idade.

```
it('Should return the default values when
   the student is not found', async () =>{

    let student =
        await studentsContract.methods
        .getEnrolledStudentByAddress(account_2)
        .call();

    assert.strictEqual(student[0], '');
    assert.strictEqual(student[1], '0');
});
```

Ao executarmos mais esse cenário observamos, novamente, no *Ganache*, o consumo de Ether, a inclusão de um bloco e os detalhes sobre mais esta transação.

Vários outros cenários podem ser escritos para nosso contrato, por exemplo, a tentativa de inclusão de dados sobre alunos partindo de um mesmo endereço — que não poderia acontecer —, a inclusão de alunos partindo de endereços diferentes — que deveria acontecer —, testes de cadastro de alunos sem informar o nome ou informando a idade como zero — que deveriam gerar uma exceção do tipo *Error*, visto estarem tratados através das diretivas `require` em nossa função principal, entre outros.

Deixamos para você praticar não apenas a geração e codificação de cenários adicionais, como também observar as alterações que nossa blockchain sofre, no *Ganache*.

7.5 Smart Contract - Telefonia celular

Tendo implementado nosso primeiro contrato podemos, agora, explorar algumas funções mais avançadas do Solidity.

Aplicaremos nosso conhecimento no desenvolvimento de um contrato utilizado por uma empresa de telefonia celular, que possui um programa de fidelidade, em que clientes — identificados por suas contas — acumulam pontos que podem ser trocados por descontos na fatura, por produtos, ou ainda, podem ser enviados para amigos que participem do programa. Além disto, nós, como donos da empresa, gostaríamos de ter acesso a funções que não devem estar disponíveis para os clientes, como a verificação do saldo do contrato, por exemplo.

Criaremos o arquivo `CellPhoneCompanyContract.sol` dentro da pasta `contracts`, ao lado de nosso `StudentsContract`, e que será usado para a implementação deste novo contrato.

Como de costume, iniciamos pelas declarações de licença, o *pragma* do compilador, e iniciamos nosso contrato, já criando nosso *struct* `Customer`, composto pelos dados de nome e saldo de pontos

— identificados pelos membros `customerName` e `customerBalance`, respectivamente.

Também criaremos o *mapping*, associando cada endereço a um usuário, tal como no contrato anterior, e já criaremos as funções de validação do nome do cliente — que não pode ser vazio —, validação do saldo — que deve ser maior ou igual a zero —, de inclusão de um cliente novo e sua consulta ao nosso *mapping*. Por ser muito similar ao que foi desenvolvido anteriormente, não entraremos em detalhes.

```
// SPDX-License-Identifier: MIT
pragma solidity = 0.8.12;

contract CellPhoneCompanyContract{

    struct Customer {
        string customerName;
        uint customerBalance;
    }

    mapping(address=>Customer) private _enrolledCustomers;

    function enrollCustomer(
        string memory _customerName
    )
        public {

            require(
                isCustomerNameValid(_customerName),
                "Name must be informed"
            );

            require(
                !isCustomerValid(
                    getEnrolledCustomerByAddress(msg.sender)
                ),
                "Customer already enrolled"
            );
```

```
        Customer memory customer;
        customer.customerName = _customerName;
        customer.customerBalance = 0;

        assert(
            isCustomerValid(customer)
        );

        _enrolledCustomers[msg.sender] = customer;
}

function getEnrolledCustomerByAddress(
    address _customerAddress
)
    public
    view
    returns (Customer memory){
        return _enrolledCustomers[_customerAddress];
}

function isCustomerValid(Customer memory _customer)
    private
    pure
    returns(bool){
        return isCustomerNameValid(
                    _customer.customerName
                )
            &&
            isCustomerBalanceValid(
                    _customer.customerBalance
                );
}

function isCustomerBalanceValid(uint _customerBalance)
    private
    pure
    returns(bool){
        return _customerBalance >= 0;
}

function isCustomerNameValid(string memory _customerName)
```

```
        private
        pure
        returns(bool){
            bytes memory tempString = bytes(_customerName);
            return tempString.length>0;
    }
}
```

Também precisamos ter uma lista dos produtos ofertados pela companhia, que poderão ser trocados pelos pontos dos clientes. Sempre que um produto for trocado por pontos, nosso contrato emitirá um `event`, que deixará a informação registrada no `log` de transações do bloco. Um evento pode ser reconhecido por uma aplicação externa e reagir a ele, porém, isso foge do escopo de nosso livro.

Inicialmente fazemos a declaração do evento `ProductExchanged`, que armazenará os dados do endereço do cliente que solicitou a troca, o índice do produto sendo trocado (desempenhando o papel de um identificador), e um indicativo de data e hora da transação. Indexaremos o endereço do cliente, por entendermos que este seja um dado de interesse para ser usado como filtro em consultas aos logs.

```
event ProductExchanged(
        address indexed _customer,
        uint _productIndex,
        uint256 _dateAndTime
    );
```

Criaremos outro *struct*, chamado `Product`, composto pelos membros `productName` (que identifica o nome do produto) e `productPoints` (indicando quantos pontos são necessários para ganhar aquele produto).

A lista de produtos é declarada como um *Array* de produtos, sem tamanho predefinido, incluído logo após a declaração de nosso *struct* `Customer`. Também, em uma variável chamada `contractOwner`, aproveitamos para armazenar o endereço do proprietário do

contrato — isto é, aquele que realizou sua publicação (que será útil mais para a frente, quando quisermos trabalhar com funções acessadas exclusivamente por ele).

```
struct Product{
    string productName;
    uint productPoints;
    uint amountExchanged;
}

address private _contractOwner;
Product[] public products;
```

Dado que queremos ter uma lista de produtos criada automaticamente, assim que o contrato for publicado, podemos, para fins didáticos, iniciá-la diretamente em seu construtor — evitando que os produtos tenham que ser armazenados manualmente, depois. Aqui, consultaremos nossa já conhecida variável global `msg.sender`, para descobrir o endereço do responsável por ter iniciado o construtor — isto é, a pessoa responsável pela publicação do contrato, e instanciaremos três produtos.

```
constructor(){

    _contractOwner = msg.sender;

    Product memory product0 =
        Product({productName: "Watch",
                productPoints: 2,
                amountExchanged:0});

    Product memory product1 =
        Product({productName: "Cellphone",
                productPoints: 5,
                amountExchanged:0});

    Product memory product2 =
        Product({productName: "Computer",
                productPoints: 10,
```

```
                    amountExchanged:0});

        products.push(product0);
        products.push(product1);
        products.push(product2);
    }
```

De acordo com o regulamento do programa de fidelidade, um cliente acumula pontos sempre que paga suas faturas; independente do valor da fatura, o cliente ganha 1 ponto sempre que realiza seu pagamento. Assim, criamos nossa função para o pagamento de contas, que será capaz de debitar da conta do usuário o valor correspondente para sua execução, e que dará um ponto para ele após o pagamento.

Aqui, pela primeira vez, usamos o modificador `payable` em uma função. Ele indica que esta função é capaz de cobrar determinada quantidade de valor da conta do cliente, caso a função seja executada com sucesso. No caso de uma exceção ocorrer, por exemplo, um *Error* advindo de uma declaração `require` no corpo da função, o valor debitado é revertido automaticamente — e aqui entendemos, na prática, o conceito das *revert functions* citadas no capítulo anterior. Uma função com esta marcação de "pagável" recebe o valor sendo transacionado, disponível na variável global `msg.value` .

```
    function payMonthlyBill(
        uint _totalDueInWei
    )
        public
        payable{

            require(
                msg.value == _totalDueInWei,
                "Total payment value is invalid"
            );

            Customer storage _customer =
               _enrolledCustomers[msg.sender];
```

```
        require(
            isCustomerValid(_customer),
            "Customer not enrolled"
        );

        _customer.customerBalance += 1;
    }
```

Um ponto a se comentar aqui foi o fato de termos buscado nosso cliente em nosso *mapping* e termos declarado sua localização como *storage*, e não como *memory*, como estávamos fazendo anteriormente.

Um *struct* no Solidity é um dado de tipo de referência. Ao fazermos sua declaração como *storage*, estamos apontando para a mesma localidade daquela armazenada no *mapping* e, portanto, quaisquer modificações realizadas dentro da função refletirão diretamente no mesmo elemento armazenado no mapeamento. Declará-lo assim evita termos que fazer a persistência no *mapping* mais uma vez ao final da função.

Podemos agora implementar a função que realiza a troca dos pontos que determinado cliente tem, por produtos da nossa lista, inicializada no construtor. Esta implementação é simples e não deveria suscitar maiores dúvidas, neste momento. Para ela, consideraremos que vamos receber, como parâmetro de entrada, um valor indicando o índice do produto dentro do nosso *Array*, representado pela variável `productIndex`.

Em seguida, buscaremos o cliente em nosso *mapping* a partir de seu endereço, buscaremos o produto a partir do índice informado, subtrairemos os pontos do cliente dos pontos requeridos pelo produto, e incrementaremos o número de vezes que o produto foi trocado — o que é uma informação estratégica interessante para o dono do contrato. Ao final, emitimos o evento `ProductExchanged`, que gerará o log de transação no bloco, contendo os dados de interesse,

e usaremos a variável global `block.timestamp` como indicativo da data e hora em que o evento foi disparado.

Como exercício, revisite essa função após o término do livro, e veja se você faria alguma alteração visando aumentar sua segurança!

Mais do que qualquer tipo de lógica de programação aqui, o mais importante neste método é que você consiga visualizar a importância das declarações `require` e `assert` incluídas no bloco de código, conforme os dados vão sendo consumidos.

```
function exchangeCustomerPointsByProduct(
    uint _productIndex
)
    public {

        require(
            _productIndex <= products.length-1,
            "Product index is not valid"
        );

        Customer storage customer =
            _enrolledCustomers[msg.sender];

        require(
            isCustomerValid(customer),
            "Customer not enrolled"
        );

        Product storage product = products[_productIndex];
        require(
            customer.customerBalance >= product.productPoints,
            "Not enough points to be used"
        );

        customer.customerBalance -= product.productPoints;
        product.amountExchanged += 1;

        assert(customer.customerBalance >= 0);
```

```
            emit ProductExchanged(
                    msg.sender,
                    _productIndex,
                    block.timestamp
                );
    }
```

Findas as implementações de funcionalidades utilizadas de forma livre pelos clientes, vamos nos focar agora na criação de funções que devem ser acessadas exclusivamente pelo administrador do contrato. Para isso, trabalharemos com a criação de um modificador personalizado, `contractOwnerOnly` que, aplicado à nossa função, impedirá que outras contas, diferentes daquela definida no construtor como dona do contrato, possam executar as funções marcadas.

A declaração de um modificador personalizado é feita utilizando a palavra reservada *modifier*, seguida de seu nome, e pela implementação do bloco de código, como demonstrado a seguir.

```
modifier contractOwnerOnly(){
    require (msg.sender == _contractOwner);
    _;
}
```

A declaração da segunda linha de código, incluindo um caractere `_`, indica que, caso a execução chegue até ali (caso ela tenha passado pela diretiva `require` com sucesso), então, a função poderá continuar a ser executada a partir daquele ponto.

Tendo o modificador sido criado, temos duas funções para implementar: a primeira, é a de consultar o saldo do contrato, isto é, o valor armazenado no membro *balance* de um dado de tipo *address*, que retorna o saldo do endereço em questão (para nós, o que interessa é o saldo do endereço do próprio contrato sendo executado, portanto, usamos a palavra reservada *this*); essa variável contém a soma dos valores pagos pelos clientes desde o início do contrato (menos qualquer retirada eventual). A segunda função será uma implementação capaz de transferir parte deste

valor para um cliente determinado, na forma de um bônus dado pela empresa, como uma estratégia de marketing.

Primeiramente, declaramos a função `getContractBalance`, que retornará o saldo disponível do contrato. Aqui, utilizaremos nosso modificador personalizado *contractOwnerOnly* em sua declaração, e acessaremos o valor de *balance* do endereço do contrato, como segue:

```
function getContractBalance()
    public
    view
    contractOwnerOnly
    returns(uint256){
        return address(this).balance;
}
```

Nossa última função é aquela capaz de transferir divisas para determinado cliente (representado por seu endereço). Ela se chamará `transferToAccount` e receberá como parâmetros um endereço válido de um cliente, e o valor a ser transferido; ela também é acessível somente pelo dono do contrato.

Aqui, também usaremos o modificador `payable`, que indica a capacidade da função de movimentar divisas, porém, vamos utilizá-lo para um endereço e não para o contrato como um todo — ao contrário da função que fazia o pagamento de contas. Mudanças recentes na linguagem desobrigam a declaração `payable` para uma variável do tipo *address*, contudo, entendemos ser uma boa prática.

Mais uma vez, por considerarmos aqui a movimentação de dinheiro, além das taxas de *Gas* cobradas naturalmente, temos a preocupação em garantir que todos os possíveis erros sejam mitigados por meio de diretivas `require` e `assert`; isto, para que, em caso de problemas no processamento, o estado seja revertido automaticamente. Sua implementação é:

```
function transferToAccount(
    address payable _destinationAddress,
```

```
        uint _amountToTransfer
    )
        public
        contractOwnerOnly
    {

        uint256 amountAvailable = address(this).balance;

        require(_amountToTransfer <= amountAvailable,
                "Balance is not enough");

        Customer memory customer =
            _enrolledCustomers[_destinationAddress];

        require(isCustomerValid(customer),
                "Destination customer is invalid");

        amountAvailable -= _amountToTransfer;

        assert(amountAvailable >= 0);

        (bool success,) = _destinationAddress
            .call{value: _amountToTransfer}("");

        require(
                success,
                "Could not transfer to destination address"
        );
    }
```

Ao contrário do que fizemos com nosso primeiro contrato, aqui não dedicaremos espaço à escrita de testes, uma vez que o foco do livro não é o trabalho com o *Mocha* ou outras bibliotecas congêneres.

No entanto, encorajo você a criar seus próprios *scripts*, realizar alterações no código para testar possibilidades e acompanhar a evolução de suas transações no *Ganache* com base nos arquivos criados anteriormente.

7.6 Criação de um Token ERC-20

Quando apresentamos o conceito de *Tokens* no Ethereum, definimos o padrão ERC-20 como aquele utilizado para a criação de *tokens* fungíveis, isto é, aqueles usados como moeda de troca para o consumo de serviços descentralizados, tal como uma ficha é utilizada em uma máquina eletrônica, por exemplo. *Tokens* desse tipo são extremamente úteis na oferta de soluções construídas com base na blockchain e, poderíamos dizer, são inclusive obrigatórios para que tenhamos projetos publicados e abertos para serem consumidos pelo público em geral.

De forma geral, um *Token* ERC-20 no Ethereum é criado quando escrevemos um contrato que implemente o padrão de dados e funções determinado pela comunidade do projeto, tal como publicado na documentação oficial do Ethereum, cujo link segue nas referências ao final do livro.

No momento da publicação deste livro, o padrão ERC-20 define a necessidade da implementação de 9 funções e 2 eventos. São eles:

```
function name()
    public view returns (string)

function symbol()
    public view returns (string)

function decimals()
    public view returns (uint8)

function totalSupply()
    public view returns (uint256)

function balanceOf(address _owner)
    public view returns (uint256 balance)

function transfer(address _to, uint256 _value)
    public returns (bool success)
```

```
function transferFrom(
    address _from,
    address _to,
    uint256 _value
)
    public returns (bool success)

function approve(
    address _spender,
    uint256 _value
)
    public returns (bool success)

function allowance(
    address _owner,
    address _spender
)
    public view returns (uint256 remaining)

event Transfer(
    address indexed _from,
    address indexed _to,
    uint256 _value
)

event Approval(
    address indexed _owner,
    address indexed _spender,
    uint256 _value
)
```

O funcionamento de cada uma dessas declarações é dado dentro do EIP-20 do Ethereum, definido em 2015, disponível na documentação do projeto. A seguir, indicamos o funcionamento de cada um deles:

1. `name()` — parâmetro opcional que indica o nome do *token*;
2. `symbol()` — parâmetro opcional que indica o símbolo usado para identificar *token*;

3. `decimals()` — parâmetro opcional que indica em quantos decimais o *token* pode ser dividido, para obter subunidades;
4. `totalSupply()` — função que retorna o total de *tokens* ofertados;
5. `balanceOf()` — função que retorna o saldo de *tokens* de determinado endereço `_owner`;
6. `transfer()` — função que realiza a transferência de um valor `_value` de *tokens*, do remetente para um endereço `_to`, mesmo que o valor de transferência seja igual a zero. Esta função obrigatoriamente deve emitir um evento `Transfer`.
7. `transferFrom()` — função que realiza a transferência de um valor `_value` entre dois endereços diferentes, não necessariamente entre o remetente e o destinatário, mesmo que o valor de transferência seja igual a zero. Esta função obrigatoriamente deve emitir um evento `Transfer`.
8. `approve()` — função intermediária, de segurança, que verifica se o `_spender` pode retirar fundos de sua conta. Caso seja possível a transação, a função deve disparar um evento `Approval`.
9. `allowance()` — função que verifica a quantidade de *tokens* que um `_spender` ainda pode retirar da conta de um `_owner`;
10. `Transfer` — evento que indica a existência de uma operação de transferência, obrigatoriamente disparado a partir das funções `transfer` e `transferFrom`;
11. `Approval` — evento que indica que, dentro de uma função `approve`, houve uma aprovação para a retirada de fundos.

Tal como um *smart contract*, o início de um arquivo de implementação do Solidity deve incluir as declarações de licença e a declaração da diretiva *pragma*. Aqui usaremos mais uma vez a licença MIT, e o compilador na versão — 0.8.12.

```
// SPDX-License-Identifier: MIT
pragma solidity = 0.8.12;
```

Como uma boa prática, declaramos o padrão de funções e eventos do ERC-20 dentro de uma `interface`. Apesar de isso não ser requerido, a razão de ser de uma `interface` é, como visto anteriormente, servir como um contrato que estipula as declarações que fazem parte do nosso *smart contract*, e sua utilização serve como mecanismo de segurança, garantindo que o padrão seja respeitado por completo. Assim, fazemos sua declaração, chamando-a de "IERC20" (o "I" ao início, indicando ser uma `interface`):

```
interface IERC20 {

    function name()
        public view returns (string);

    function symbol()
        public view returns (string);

    function decimals()
        public view returns (uint8);

    function totalSupply()
        public view returns (uint256);

    function balanceOf(address _owner)
        public view returns (uint256 balance);

    function transfer(address _to, uint256 _value)
        public returns (bool success);

    function transferFrom(
        address _from,
        address _to,
        uint256 _value
    )
        public returns (bool success);
```

```
    function approve(
        address _spender,
        uint256 _value
    )
        public returns (bool success);

    function allowance(
        address _owner,
        address _spender
    )
        public view returns (uint256 remaining);

    event Transfer(
        address indexed _from,
        address indexed _to,
        uint256 _value
    );

    event Approval(
        address indexed _owner,
        address indexed _spender,
        uint256 _value
    );

}
```

Em seguida, fazemos a implementação de nosso contrato (de nome "TokenERC20", mas que poderia receber qualquer outro nome), indicando que ele implementa a interface `IERC20`, através da palavra reservada `is`:

```
contract TokenERC20 is IERC20 {

}
```

Seguimos com a implementação do nosso *Token*, ao que vamos chamar de "CasadoCodigo", e que daremos o símbolo "CDC"; por ser um *token* do Ethereum, consideraremos que o CDC se divida em 18 decimais também, exatamente como um Ether, subdividido em Weis. Aqui, podemos fazer essas declarações como constantes

dentro do contrato ou poderíamos criar funções que retornassem esses valores; ficamos com a primeira opção, considerando a limpeza do código.

Para esta implementação, vamos nos basear, ainda, no padrão de implementação utilizado pelo OpenZeppelin, um padrão de segurança bastante popular utilizado por desenvolvedores de contratos que é, portanto, uma implementação que já considera as boas práticas.

```
string public constant name = "CasadoCodigo";
string public constant symbol = "CDC";
uint8 public constant decimals = 18;
```

Nossa propriedade `totalSupply` não pode ser uma constante, visto ser um valor que vai ser alterado o tempo todo, conforme novas transações utilizem nosso *token*. Assim, fazemos sua declaração como um *uint256* precedido de um símbolo *underscore*, indicando ser uma variável de uso interno do contrato.

Aqui, aproveitamos para criar dois *mappings* também internos:

1. O primeiro deles, chamado `_balances`, controlará o balanço monetário para determinado endereço, controlando quantos *tokens* o remetente possui para uso, e que será usado nas implementações das funções `balanceOf`, `transfer` e `transferFrom`.

2. O segundo é um *mapping* de um *mapping*, chamado `_allowed`, e que guarda os valores previamente aprovados para serem transferidos entre endereços diferentes (o *owner*, o possuidor da quantia, e o *spender*, a pessoa que está consumindo os *tokens*). Ele será utilizado para a implementação das funções `approve`, `allowance` e `transferFrom`.

Por fim, podemos aproveitar e declarar nosso construtor, que se limitará a associar o valor do `totalSupply` (que, por não possuir valor

declarado, possui seu valor-padrão — no caso de um `uint256`, zero), à chave representada pelo endereço do remetente.

Estas declarações, em nosso contrato, ficam:

```
uint256 private _totalSupply;
mapping(address => uint256) private _balances;
mapping(address => mapping (address => uint256))
    private _allowed;

constructor() {
    balances[msg.sender] = totalSupply_;
}
```

Seguimos com a declaração das funções `totalSupply()`, `balanceOf(address tokenOwner)` e `allowance(address owner, address delegate)`, que se limitam a retornarem os valores das nossas variáveis declaradas anteriormente. Como estes são valores dinâmicos, não puderam ser declarados como constantes. Reparem, ainda, no funcionamento dos *mappings*, de acordo com os endereços usados como parâmetros nestas funções.

```
function totalSupply()
    public override view returns (uint256) {
    return _totalSupply;
}

function balanceOf(address _owner)
    public override view returns (uint256) {
        return _balances[_owner];
}

function allowance(address _owner, address _spender)
    public override view returns (uint) {
        return _allowed[_owner][_spender];
}
```

Nossa implementação da função `transfer` será bastante simples, e será composta por 5 passos:

1. Verificar, através de uma diretiva `require`, que o valor solicitado para transferência seja menor ou igual ao saldo disponível para o remetente;
2. Verificar, através de uma diretiva `require`, que o endereço informado como parâmetro é diferente do chamado "endereço zero", isto é, o valor *default* para uma variável `address`, sendo, portanto, um endereço válido que se refira a uma conta efetiva;
3. Fazer subtração do valor a ser retirado, do saldo disponível para o endereço do remetente;
4. Fazer adição do valor retirado no passo anterior à conta do destinatário;
5. Disparar o evento `Transfer`.

A implementação destes passos é a seguinte:

```
  function transfer(address _to, uint256 _value)
      public returns (bool) {
          require(
              _value <= _balances[msg.sender],
              "Not enough funds!"
          );

          require(
              _to != address(0),
              "Address is invalid"
          );

          _balances[msg.sender] =
              _balances[msg.sender] - _value;

          _balances[_to] = _balances[_to] + _value;
          emit Transfer(msg.sender, _to, _value);
          return true;
  }
```

Nossa função `approve` se limitará a aprovar qualquer valor que tenha sido passado como parâmetro, sem outras regras de negócio, que podem ser trabalhadas no contrato principal que utilize o *token*. Ela fará uma verificação de segurança, através de uma diretiva `require`, que o valor do endereço não é um valor *default* e, em seguida, atualizará o *mapping* `_allowed` com o valor recebido. Ao final, emitirá um evento `Approval`. Sua implementação é a seguinte:

```
    function approve(
        address _spender,
        uint256 _value
    )
        public returns (bool) {

            require(
                _spender != address(0),
                "Address is invalid"
            );

            _allowed[msg.sender][_spender] = _value;
            emit Approval(msg.sender, _spender, _value);
            return true;
  }
```

Finalmente, para a implementação da função `transferFrom`, seguiremos a mesma estratégia adotada para a função `transfer`, porém, agora, incluiremos uma verificação intermediária para garantir que o valor solicitado também é menor ou igual ao valor previamente aprovado para consumo, ajustado por meio da função `approve`, através da inclusão de uma diretiva `require`. Ao final, subtrairemos o valor transferido do valor disponível no mapping `_allowed`, que passa a indicar agora um valor menor do que aquele pré-aprovado.

```
   function transferFrom(
        address _from,
        address _to,
        uint256 _value
    )
```

```
public
returns (bool)
  {
    require(
        _value <= _balances[_from],
        "Not enough funds!"
    );

    require(
        _value <= _allowed[_from][msg.sender],
        "Not enough allowed funds"
    );

    require(
        _to != address(0),
        "Address is invalid"
    );

    _balances[_from] = _balances[_from] - _value;
    _balances[_to] = _balances[_to] + _value;

    _allowed[_from][msg.sender] =
        _allowed[_from][msg.sender] - _value;

    emit Transfer(_from, _to, _value);
    return true;
  }
```

7.7 Criação de um Token ERC-721 (NFT)

O padrão ERC-721 surge no Ethereum, poucos anos após a definição do EIP-20, quando o padrão ERC-20 já estava ativo. Esse padrão foi criado de forma similar ao ERC-20, e com base nele, e define a necessidade da implementação de 9 funções e 3 eventos.

Diferentemente do padrão ERC-20, no entanto, o trabalho com *tokens* ERC-721 pode exigir o trabalho com outros padrões para

serem consumidos, como é o caso, por exemplo, do ERC-165, que deve ser implementado por contratos que queiram consumir *tokens* deste tipo. Existem ainda implementações muito específicas, requeridas, por exemplo, por provedores de carteiras, ou ainda, implementações de enumerações que permitam ao NFT ser listado em aplicações que o consumam. Aqui, vamos nos limitar a apresentar a implementação do padrão por meio do Solidity.

Tal como apresentado na seção anterior, iniciaremos a implementação do nosso *token* pelas declarações de licença, da diretiva *pragma*, e pela criação de uma interface, que enumera todas as funções e eventos requeridos pelo padrão.

```
// SPDX-License-Identifier: MIT
pragma solidity = 0.8.12;

interface IERC721 {
    function balanceOf(address _owner)
        external view returns (uint256);

    function ownerOf(uint256 _tokenId)
        external view returns (address);

    function safeTransferFrom(
        address _from,
        address _to,
        uint256 _tokenId,
        bytes data
    ) external payable;

    function safeTransferFrom(
        address _from,
        address _to,
        uint256 _tokenId
    ) external payable;

    function transferFrom(
        address _from,
        address _to,
        uint256 _tokenId
```

```
    ) external payable;

    function approve(
        address _approved,
        uint256 _tokenId
    ) external payable;

    function setApprovalForAll(
        address _operator,
        bool _approved
    ) external;

    function getApproved(uint256 _tokenId)
        external view returns (address);

    function isApprovedForAll(
        address _owner,
        address _operator
    ) external view returns (bool);

    event Transfer(
        address indexed _from,
        address indexed _to,
        uint256 indexed _tokenId
    );

    event Approval(
        address indexed _owner,
        address indexed _approved,
        uint256 indexed _tokenId
    );

    event ApprovalForAll(
        address indexed _owner,
        address indexed _operator,
        bool _approved
    );
}
```

O funcionamento dessas funções é descrito no EIP-721 do Ethereum, definido em 2018, e disponível na documentação do

projeto (cujo endereço está disponível nas referências). Em resumo, elas representam:

1. `balanceOf()` — retorna a quantidade de *tokens* que determinado endereço possui associados;
2. `ownerOf()` — função que retorna o endereço definido como o "possuidor" do *token*, representado por seu `_tokenId` ;
3. `safeTransferFrom()` — função definida duas vezes, com números de parâmetros variáveis, usada para transferir a posse de um *token* entre endereços distintos, de forma segura, garantindo que tanto o remetente como o destinatário sejam contratos e que estejam prontos para trabalhar com o padrão ERC-721;
4. `transferFrom()` — função que realiza a transferência da posse de um *token* entre endereços e, ao final, emite um evento `Transfer` ;
5. `approve()` — função que aprova o endereço `_approved` a operar sobre determinado *token*, representado por seu `_tokenId` ;
6. `setApprovalForAll()` — função que dá permissão a um `_operador` a atuar sobre todos os *tokens* de posse de um endereço `_owner` ;
7. `getApproved()` — função que retorna o endereço aprovado a operar sobre determinado *token*, representado por seu `_tokenId` ;
8. `isApprovedForAll()` — função que verifica se um endereço `_owner` possui aprovação para trabalhar sobre um segundo endereço `_operator` ;
9. `Transfer` — evento que indica a existência de uma operação de transferência;
10. `Approval` — evento que indica que, dentro de uma função `approve` , houve uma aprovação para a operação sobre

determinado *token*, representado por seu `_tokenId` ;

11. `ApprovalForAll` — evento que indica que, dentro de uma função `setApprovalForAll` , um `_operador` recebeu permissão para atuar sobre todos os _tokens de posse do endereço do `_owner` .

Iniciamos a declaração do nosso contrato `TokenERC721` , indicando que ele implementa a interface `IERC721` , através da palavra reservada `is` .

```
contract TokenERC721 is IERC721 {

}
```

Aqui, trabalharemos com quatro variáveis do tipo *mapping*, a que chamaremos `_owners` , `_balances` , `_tokenApprovals` e `_operatorApprovals` , que associam, respectivamente, um *token* (representado pelo seu `tokenId`) a um endereço (indicando propriedade), um endereço ao número de *tokens* associados a ele, *tokens* (representados por seus *tokenIds*) a endereços aprovados para operarem sobre eles, e endereços a um outro *mapping*, indicando os endereços sobre os quais o primeiro endereço possui permissão para operar. Suas implementações são dadas a seguir:

```
mapping(uint256 => address) private _owners;

mapping(address => uint256) private _balances;

mapping(uint256 => address) private _tokenApprovals;

mapping(address => mapping(address => bool))
    private _operatorApprovals;
```

Em seguida, começamos a implementar as funções definidas na *interface*. Inicialmente, implementaremos a função `balanceOf` , que se limita a verificar se o endereço recebido como parâmetro é válido (isto é, seu valor é diferente do valor *default*, comparado através da diretiva `address(0)`) e, em seguida, retorna a quantidade de *tokens* associados àquele endereço, em consulta direta ao *mapping*.

```
 function balanceOf(address _owner)
    public view virtual override returns (uint256) {
        require(
            _owner != address(0),
            "Address is invalid"
        );

        return _balances[_owner];
}
```

A próxima função implementada, a `ownerOf` recebe como parâmetro um `_tokenId` e consulta no mapping `_owners`, quem é o endereço associado ao ativo em questão. Como forma de segurança, incluímos uma diretiva `require` após a consulta, para garantir que o endereço consultado seja válido (não possua o valor *default*)

```
function ownerOf(uint256 _tokenId)
    public view virtual override returns (address) {
        address owner = _owners[_tokenId];
        require(
            owner != address(0),
            "Address is invalid"
        );
        return owner;
    }
```

Em seguida, podemos declarar algumas funções auxiliares, que nos retornam os valores de estado nos *mappings* iniciais. A primeira delas, a `isApprovedForAll` retorna o valor encontrado no *mapping* `_operatorApprovals`, utilizando a combinação de chaves do endereço do proprietário e o endereço do operador, ambos recebidos via parâmetro. Sua implementação é:

```
function isApprovedForAll(
    address _owner,
    address _operator
) public view virtual override returns (bool) {
        return _operatorApprovals[_owner][_operator];
    }
```

Logo abaixo, declaramos a função auxiliar `getApproved`, que tem o objetivo de retornar o endereço cadastrado no *mapping* `_tokenApprovals` como um valor referente à chave indicada pelo `tokenId`. Em outras palavras: ela indica qual endereço tem aprovação para manipular determinado *token*. Como etapa de segurança, utilizamos uma diretiva `require` para garantir que o *token* possua pelo menos um proprietário (isso é conseguido garantindo que o endereço associado não seja o valor *default* para uma variável *address*)

```
function getApproved(uint256 _tokenId)
    public view virtual override returns (address) {
        require(
            _owners[tokenId] != address(0),
            "Token is not owned by anyone"
        );

        return _tokenApprovals[_tokenId];
}
```

A terceira função de aprovação, `approve`, é aquela capaz de associar o valor do endereço de destino ao *mapping* `_tokenApprovals`, indicando que aquele *token* (representado por seu `tokenId`) está aprovado para ser trabalhado pelo novo endereço. Aqui, iniciamos o fluxo com a utilização de duas diretivas `require`, que garantem tanto que o endereço de destino e o de origem sejam diferentes como que o remetente seja também o proprietário do *token* (ou que está aprovado para trabalhá-lo); ao final, um evento `Approval` é emitido. Sua implementação é dada a seguir:

```
function approve(
    address _to,
    uint256 _tokenId
) public virtual override {

    address owner = ownerOf(_tokenId);

    require(
```

```
        _to != owner,
        "Owner must be different from destiny"
    );

    require(
        msg.sender == owner ||
            isApprovedForAll(owner, msg.sender),
        "Sender is not owner or token is not approved"
    );

    _tokenApprovals[tokenId] = _to;
    emit Approval(ownerOf(_tokenId), _to, _tokenId);
}
```

Finalmente, a função `setApprovalForAll` tem como objetivo atualizar o *mapping* `_operatorApprovals`, ajustando o valor correspondente ao endereço de origem para a chave do endereço de operação `_operator` igual a `true` ou `false`, dependendo da operação desejada — sendo o valor `true` utilizado quando se deseja aprovar, e `false` quando se deseja bloquear a aprovação — emitindo, ao final, um evento `ApprovalForAll`. Antes disto, apenas utilizamos uma diretiva `require` para garantir que o endereço de origem seja diferente do endereço do operador, o que não faria sentido para o funcionamento de nosso *token*.

```
function setApprovalForAll(
    address _operator,
    bool _approved
) public virtual override {
    require(
        msg.sender != _operator,
        "Sender must be different from operator"
    );

    _operatorApprovals[msg.sender][_operator] =
        _approved;

    emit ApprovalForAll(msg.sender, _operator, _approved);
}
```

Para implementar as funções de transferência, criaremos duas funções internas, chamadas `_transfer` e `_safeTransfer`, que serão invocadas por meio das funções principais, e que permitirão que reaproveitemos parte do código.

Começamos pela definição da função `_transfer`, que recebe como parâmetros os endereços de origem e destino, e o `tokenId`, representando o *token* em contexto. Inicialmente, através de diretivas `require`, a função verifica que o endereço de origem da solicitação seja o proprietário do *token*, consultando na função `ownerOf`, declarada anteriormente e, em seguida, garante que o endereço de destino não seja um endereço com valor *default*.

Em seguida, ela remove a permissão de trabalho do endereço sobre o *token*, através da chamada à função interna `_approve`, definindo como o endereço aprovado para atuar sobre o *token* o endereço *default*. Na prática, aqui, retiramos qualquer permissão de trabalho em cima do *token* em questão.

Por fim, alteramos os *mappings* `_balance` e `_owners`, decrementando o valor de contagem do endereço de origem em 1 unidade, incrementando o valor de contagem do endereço de destino em 1 unidade, e associando o endereço de destino como o novo proprietário do *token* representado pelo `tokenId`, emitindo, ao final, um evento `Transfer`.

```
    function _transfer(
        address _from,
        address _to,
        uint256 _tokenId
    ) internal virtual {
        require(
            ownerOf(_tokenId) == _from,
            "Token is not owned by origin"
        );

        require(
            _to != address(0),
```

```
            "Address is invalid"
        );

        _approve(address(0), _tokenId);

        _balances[_from] -= 1;
        _balances[_to] += 1;
        _owners[_tokenId] = _to;

        emit Transfer(_from, _to, _tokenId);
    }
```

Com fins de simplificação, nossa função `_safeTransfer` se limitará a verificar se o endereço de destino é um outro contrato, e não mais do que isso. Em um caso real, precisaríamos verificar também se, além de ser um contrato, ele é um contrato preparado para trabalhar com *tokens* ERC-721 (para isso, trabalharíamos com o parâmetro `_data` , recebido), o que vai de encontro com o que mencionamos no início desta seção: que, no trabalho com NFTs, existem pré-requisitos que precisam ser preenchidos por diversos atores. Sendo o destino um contrato, invocamos nossa função interna `_transfer` , e seguimos com o fluxo normal de transferência.

```
    function _safeTransfer(
        address _from,
        address _to,
        uint256 _tokenId,
        bytes memory _data
    ) internal virtual {
        require(
            _to.isContract(),
            "Destiny is not a contract"
        );

        _transfer(_from, _to, _tokenId);
    }
```

Finalmente podemos declarar as implementações das nossas funções de transferência `safeTransferFrom` e `transferFrom` , todas

trabalhando por meio das invocações das funções internas `_transfer` e `_safeTransfer`.

Apenas como precaução adicional, criaremos uma função interna `_tokenIsOwnedOrApproved` que será utilizada em diretivas `require` e que garantirá que o *token* em questão existe, e que ele é de propriedade do endereço de origem ou que este endereço possua permissão para trabalhar com ele (tenha sido aprovado).

```
    function _tokenIsOwnedOrApproved(
        address _spender,
        uint256 _tokenId
    ) internal view virtual returns (bool) {

        require(
            _exists(_tokenId),
            "Token not found"
        );

        address owner = ownerOf(_tokenId);

        require(
            _spender == owner ||
                getApproved(_tokenId) == _spender ||
                isApprovedForAll(owner, _spender),
            "Origin is not the owner or is not approved"
        );

        return true;
    }
```

As implementações das funções `safeTransferFrom` e `transferFrom` são finalmente possíveis.

```
    function safeTransferFrom(
        address _from,
        address _to,
        uint256 _tokenId
    ) public virtual override {
        safeTransferFrom(_from, _to, _tokenId, "");
```

```
}

function safeTransferFrom(
    address _from,
    address _to,
    uint256 _tokenId,
    bytes memory _data
) public virtual override {

    require(
        _tokenIsOwnedOrApproved(msg.sender, _tokenId),
        "Origin is not the owner or is not approved"
    );

    _safeTransfer(from, to, tokenId, _data);
}


function transferFrom(
    address _from,
    address _to,
    uint256 _tokenId
) public virtual override {

    require(
        _tokenIsOwnedOrApproved(msg.sender, _tokenId),
        "Origin is not the owner or is not approved"
    );

    _transfer(_from, _to, _tokenId);
}
```

7.8 Otimizando os custos de transação

Se por um lado a sintaxe do Solidity é bastante simples e fácil de ser aprendida, principalmente por pessoas que já tenham experiência com outras linguagens de programação, grande parte do valor de

quem desenvolve *smart contracts* advém de sua capacidade de otimizar os custos de execução.

Contratos inteligentes precisam ser publicados e executados na EVM, e isso incorre no pagamento de taxas na forma de *Gas*. Quando uma desenvolvedora se mostra capaz de diminuir os custos associados a seus contratos, isso é um indicativo de que ela conhece bem a estrutura sobre a qual a linguagem foi montada, entendendo detalhes de sua arquitetura, que são fundamentais para sua maior eficiência.

O primeiro ponto a ser analisado por desenvolvedores nesta tarefa diz respeito à revisão das variáveis utilizadas e seus tipos. O Solidity trabalha com um tamanho de palavra igual a 256 bits (32 bytes), e as variáveis declaradas são “empacotadas” em *slots* desse tamanho, na ordem em que são declaradas. Assim, a declaração de três variáveis na ordem *uint8*, *uint256*, *uint8* torna-se computacionalmente mais cara do que a declaração de variáveis *uint8*, *uint8* e *uin256* — este gastando dois slots, enquanto a primeira forma gasta três, já que os espaços das duas primeiras variáveis se somam, ocupando menos do que um slot.

A atribuição de valores iniciais para variáveis utiliza recursos computacionais e só deveria ser feita em caso de necessidade. Por exemplo, todas as variáveis no Solidity possuem um valor padrão, seja o zero, para números inteiros, seja o valor vazio para Strings. Assim, uma atribuição de inicialização do tipo `uint variavel = 0` é mais cara do que a mera declaração `uint variavel`.

No caso de uma variável que já possua um valor e que precise ser reinicializada, o Solidity possui uma declaração `delete`, cujo uso deve ser encorajado.

Ainda considerando a questão do empacotamento, há de se lembrar que existe um gasto computacional ao adequar um elemento de tamanho menor em um *slot* de 256 bits. Quando tratamos de variáveis isoladas, únicas, por exemplo, do tipo *uint8* (ou um

tamanho menor que 256), elas podem ter um custo em *Gas* maior do que a declaração de uma variável isolada *uint256*, que se adéqua perfeitamente ao *slot*. E isso também tem que ser pensado, justamente por isso é que em nosso primeiro *smart contract* declaramos a idade como um *uint* e não como um *uint8*, que seria adequado para trabalhar com idades. Além disso, alguns tipos de dados não podem ser empacotados; é o caso, por exemplo, de elementos em memória e *mappings*.

Structs e *Arrays* sempre iniciam um novo *slot*, independentemente do tamanho disponível e são, portanto, normalmente mais caros, ainda que seus membros sejam armazenados separadamente, depois. Se estamos trabalhando com tipos muito simples, eventualmente um *Array* pode sair mais barato do que um *mapping*, por exemplo, já que seus membros são empacotados em conjunto. *Arrays* com tamanhos fixos são mais baratos que os de tamanho dinâmico, logo, também deveriam ser preferidos.

Operações em variáveis em memória são mais baratas do que operações em variáveis localizadas no *storage*. Assim, uma boa tática é manipular uma variável em memória o quanto seja necessário para apenas então enviar seus valores para uma variável em *storage*.

Um importante fator que influencia os custos de *Gas* associados ao contrato, o tamanho do *bytecode* é resultado direto do tamanho do código escrito. Assim, no Solidity, boas práticas comuns em linguagens de programação, como a quebra de funções em funcionalidades pequenas com escopo único, são mais caras do que a escrita de funcionalidades com grande complexidade, que trazem as contrapartidas comuns do desenvolvimento de software — e, portanto, há um balanço a ser estudado.

O uso de bibliotecas externas diminui o tamanho do *bytecode*, porém, traz a contrapartida de associar custos computacionais de chamadas externas.

Ainda no âmbito das funções, chamadas feitas para funções internas são geralmente mais baratas que chamadas feitas para funções públicas, o que é mais um fator de decisão, que envolvem as boas práticas de desenvolvimento.

Como último recurso, o compilador do Solidity possui ele próprio um otimizador de código, que trabalha no *assembly*, separando sequências de instruções em blocos, a partir de regras de simplificação do próprio compilador. Desta forma, expressões complexas são transformadas em versões mais simples, com menor custo computacional e mais eficientes que, eventualmente, também são capazes de reduzir o código — evitando, por exemplo, a existência de duplicações. A contrapartida de sua utilização é que os tempos de compilação são maiores e, portanto, pode se tornar um empecilho em tempo de desenvolvimento.

Conclusão

Neste capítulo finalizamos a segunda parte deste livro, que se propôs a apresentar as ferramentas para o desenvolvimento de aplicações que utilizem a blockchain como base de funcionamento. Aqui, pudemos colocar em prática o conteúdo teórico discutido anteriormente neste livro, e alguns conceitos um pouco mais "abstratos" foram colocados em prática, dando uma ideia verdadeira de como a linguagem Solidity é utilizada em projetos reais. A execução dos testes unitários em cima do *smart contract* possibilitou, ainda, acompanhar o funcionamento da blockchain durante as etapas de publicação e consumo do contrato, este, realizado através de transações.

Em seguida, fizemos uma implementação dos dois principais padrões de *tokens* utilizados pelo Ethereum, o ERC-20 e o ERC-721 que versam, respectivamente, sobre os padrões de *tokens* fungíveis e *tokens* infungíveis (NFTs), ambos tipos de aplicações cobiçadas pelos mercados — que estão em forte expansão —, para os quais a demanda por profissionais capacitados certamente traz boas oportunidades para o profissional que conheça seus

funcionamentos. Aqui demonstramos, de fato, como o Solidity é uma ferramenta poderosa, capaz de atuar nos diversos contextos de negócios existentes.

Por fim, dedicamos alguns parágrafos à tarefa de otimização de custos, trabalho que reflete diretamente o conhecimento dos desenvolvedores, uma vez que demanda conhecimentos que vão além da sintaxe, em que decisões baseadas na arquitetura da linguagem podem fazer a diferença nos custos de publicação e consumo. Por refletirem diretamente nos custos associados ao projeto, essas decisões são fundamentais para garantir seu sucesso e, portanto, fazem parte de uma etapa importante do desenvolvimento dos *smart contracts*.

Nos próximos capítulos aumentaremos nossas ferramentas de trabalho em projetos para o Ethereum. Discutiremos temas de segurança, que lidam diretamente com questões de vulnerabilidades de contratos — questões extremamente oportunas em um ambiente de trabalho em que nossos códigos são imutáveis e que precisam estar em conformidade desde o primeiro instante. Terminaremos a última parte do livro discutindo questões de armazenamento na blockchain, e apresentando uma frente de trabalho que começa a ganhar fôlego no mundo descentralizado: o trabalho com dados e *analytics*.

Parte 3 - SEGURANÇA E ARMAZENAMENTO NA BLOCKCHAIN ETHEREUM

Capítulo 8
Segurança na Rede Ethereum

8.1 Introdução

Apesar de ser uma tecnologia segura por natureza, a blockchain, assim como qualquer sistema computacional, possui vulnerabilidades e propensões a ataques que devem ser minimamente conhecidos e endereçados pela pessoa desenvolvedora. Além de movimentar dinheiro como transações, a tecnologia blockchain, quando aplicada a um contexto estratégico, está ligada diretamente ao negócio para o qual ela foi construída, tornando-a um ponto sensível em toda a cadeia.

Some-se a isso o fato de os *smart contracts* serem imutáveis, o que acende um sinal de alerta importante e que exige que seu desenvolvimento seja feito de forma segura. Em capítulos anteriores discutimos sobre o ataque à DAO que, ainda que feito maliciosamente — e, portanto, de idoneidade questionável — resumiu-se ao consumo de um *smart contract*, que apresentava uma vulnerabilidade e que era de conhecimento público, fato este que culminou com a cisão do projeto Ethereum.

Existe hoje um esforço por parte de organizações para minimamente catalogar os tipos de vulnerabilidades e ataques que possam ser aplicados a uma rede blockchain. Ainda que alguns sejam muito conhecidos desde as primeiras implementações de cadeias de blocos, outros foram "descobertos" mais recentemente;

dentre eles, existem ataques para os quais não se conhecem sequer soluções de fácil implementação. Um exemplo é o chamado “Ataque de 51%”, que explora os mecanismos de consenso, e que é responsável pelo constante aumento na dificuldade de solução dos quebra-cabeças lógicos — e isso, no Ethereum, como já visto, tem trazido efeitos colaterais importantes.

Em geral, a segurança de uma rede blockchain é diretamente proporcional ao seu poder de computação de *hashes*. Sua própria natureza apresenta aqui um concorrente que deverá receber atenção nos próximos anos: a *computação quântica*, que, em teoria, seria capaz de realizar ataques de força bruta com eficiência — que coloca em risco não apenas a blockchain, mas todo um ecossistema de segurança da informação implementado em diferentes níveis.

Nos próximos tópicos discutiremos algumas vulnerabilidades encontradas na arquitetura blockchain, comuns a diversas redes e, em seguida, focaremos nas fragilidades encontradas durante o desenvolvimento de *smart contracts* que podem ser exploradas maliciosamente.

Por fim, discutiremos algumas das opções de trabalho seguro e enumeraremos boas práticas de segurança a serem aplicadas no trabalho em nossa rede.

8.2 Redes de tolerância a Falha Bizantinas

Dentro do contexto de sistemas computacionais distribuídos, um tipo de situação bem conhecido são as chamadas *Falhas Bizantinas*, suscetíveis a ocorrerem quando pelo menos um dos componentes do sistema apresenta algum tipo de falha e os demais precisam decidir sobre qual ação tomar. Sendo a blockchain uma rede P2P,

distribuída, é também um ambiente em que este tipo de ocorrência pode surgir.

O nome “Falha Bizantina” advém de um dilema lógico chamado de "Problema dos Generais Bizantinos", que debate sobre os meios de comunicação e a decisão que generais de um exército devem tomar para entrarem em um consenso sobre o próximo movimento de guerra, e que leva em conta a possibilidade de algum deles mentir ou ter outros interesses (particulares) e que este seja capaz de persuadir os demais a decidirem apoiá-lo em determinada decisão. Esse dilema ilustra bem o funcionamento de uma blockchain quando são aplicados os algoritmos de consenso, em que cada nó da rede P2P deve concordar sobre o real estado da rede, sendo o sistema capaz de identificar ações maliciosas e revertê-las.

Em uma maneira ilustrativa, o dilema considera a existência de quatro generais de um exército (bizantino) que estão cercando uma fortaleza e, agora, precisam decidir se devem atacá-la ou não. O mais intuitivo aqui seria considerar que uma decisão da maioria deveria decidir ou não pelo ataque — portanto, ao menos, por três votos favoráveis.

O problema surge quando se considera a possibilidade da existência de ao menos um impostor (dentro destes três votos a favor), cujo voto não representa necessariamente a melhor escolha, mas que traria uma vantagem para ele. Na melhor das hipóteses (considerando um único impostor), isso já baixaria o total dos três votos válidos para dois, o que representa um empate e, portanto, invalidaria a tomada de decisão.

Esse problema foi apresentado formalmente em publicação de 1982 por Leslie Lamport, Robert Shostak e Marshal Pease, denominada *The Byzantine Generals Problem* (link segue nas referências). Nela, os pesquisadores debatem seu funcionamento considerando o caso de que cada general recebe a informação e a repassa oralmente para os demais, considerando a possibilidade da presença de

impostores. Apesar de parecer um problema de solução simples, sua formalização é bastante complexa.

Considerando o exemplo dos quatro generais, o algoritmo de resolução proposto funciona quando uma ordem é recebida e repassada para os demais, e há um consenso de ao menos dois terços dos participantes. De forma didática, e simplificada, ele funciona assim:

1. Em um cenário em que existam quatro generais, um deles dá uma primeira ordem para o restante (atacar ou retroceder);
2. Cada um dos generais repassa a informação para todos os outros — logo, cada general envia a ordem para outros três generais;
3. Se um impostor mentir sobre a ordem recebida, ainda assim, as outras duas ordens recebidas serão a maioria e, portanto, elas é que pesarão na decisão.

A resolução funciona muito bem para condições bem estabelecidas; entre elas, o fato de considerar um único impostor dentro dos quatro generais. Caso esse número fosse maior, o consenso penderia para o lado incorreto e é justamente essa brecha que origina alguns dos ataques conhecidos da blockchain, que serão vistos a seguir.

8.3 Ataques de 34% e 51%

Os ataques de 34% e 51% são dois tipos de ataques perpetrados contra cadeias blockchain e que se baseiam na existência de uma “maioria maliciosa" capaz de manipular o resultado do consenso real, esperado pelos participantes da rede.

O primeiro deles é simples de ser entendido considerando os passos da solução do Problema dos Generais Bizantinos. O algoritmo em questão considera a existência de um único general malicioso, funcionando bem, portanto, para uma troca de

mensagens orais com dois terços de certeza (ou, visto ao contrário, com no máximo um terço de falhas). Caso tivéssemos dois generais maliciosos — ou, mais corretamente, mais de um terço das mensagens sendo transmitidas incorretamente — a maioria não seria atingida e, portanto, o consenso penderia para o lado incorreto. O número mágico de 34% advém daí: qualquer número de mensagens incorretas maior do que um terço (aproximadamente 33%) tenderia a formar um consenso inadequado sobre a inclusão de um bloco na rede.

O ataque de 51% é mais elaborado e também provê ao agente malicioso um controle ainda maior. Ele é um tipo de ataque de infiltração na rede que provê ao atacante o controle sobre toda a cadeia, ao contrário do ataque de 34%, que tem efeitos localizados.

Em um cenário propício para a ocorrência do ataque de 51%, um grupo de mineradores/validadores maliciosos com poder computacional maior do que o restante da rede — e daí o nome 51%, denotando a maioria do poder computacional disponível — começa a participar no processo de validação das transações. Por possuir maior poder de cálculo, eventualmente, esse grupo passará a ser o único capaz de incluir novos blocos à cadeia, monopolizando o processo de inclusão de transações.

Ao incluírem novos blocos à rede, os participantes são recompensados com uma determinada quantia em criptomoedas, o que é válido para cada bloco adicionado e confirmado. Até então, nada de ilegal está ocorrendo.

O que acontece é que, uma vez que os participantes maliciosos, que controlam as transações incluídas nos blocos, são remunerados, eles podem movimentar suas divisas na compra de bens ou no câmbio por outras moedas — mesmo moedas físicas, como o dólar americano. Ao mesmo tempo, são capazes de impedir que essas novas transações sejam incluídas no bloco, criando a ilusão de que eles ainda possuem o dinheiro inicial, em

criptomoedas, visto que as transações de sua movimentação não foram incluídas nos blocos recentes.

Este fenômeno é conhecido pela expressão em inglês *double spend*, traduzida ao pé da letra como “gasto em dobro”, denotando a capacidade que os atores maliciosos têm de utilizar suas divisas mais de uma vez.

Esse tipo de ataque é mais difícil de ocorrer em grandes blockchains como o Bitcoin ou o Ethereum, visto que o número de participantes nessas redes é muito grande e, alcançar um poder computacional maior do que a soma de todos os poderes computacionais de todos os participantes atuais das redes é uma tarefa difícil. No entanto, isso já ocorreu no passado, em ambas as redes. Mais recentemente, o Ethereum Classic passou por uma série de ataques desse tipo entre 2019 e 2020, com prejuízos financeiros alcançando os 5 milhões de dólares em um deles.

Infelizmente não existe uma fórmula mágica para a proteção da rede contra esse tipo de uso, e apenas o sucesso da blockchain é que pode determinar sua vulnerabilidade a esse ataque. Outro ponto de atenção é o de que, visto ser um ataque planejado e executado por mineradores, ele pode ser usado como ferramenta política de negociação e ameaça.

Mudanças recentes nas políticas de remuneração da Rede Ethereum, em 2021, e a rápida migração para a versão do Ethereum 2.0 (que terminará diminuindo os ganhos financeiros obtidos na etapa de mineração), trouxeram uma grande insatisfação para parte dos mineradores. Alguns deles passaram a fazer ameaças de “demonstração de força” contra a rede — ameaçando, eles mesmos, se unirem para controlarem a rede caso suas demandas não sejam atendidas.

8.4 Ataques de corrida e de Finney

Ataques de corrida são um outro tipo de execução maliciosa, que culminam com o *double spend* na rede. Nesse tipo de ação, agentes maliciosos são capazes de gerar uma transação maliciosa, não confirmada, e dão a impressão de incluí-la no nó da vítima, dando a falsa impressão, para ela, de que todo o processo foi concluído com sucesso, induzindo-a ao erro.

Uma vez que o ataque é bem localizado, e que o restante da rede entenda que aquela transação é inválida, ela acabará por ser desfeita e removida da cadeia, porém, a uma altura em que a vítima já tenha tido prejuízos financeiros.

Uma variação desse tipo de ataque, típica de um ecossistema de blockchain, é o Ataque de Finney — nome dado em homenagem a Hal Finney, a primeira pessoa a utilizar um Bitcoin na história.

O Ataque de Finney compreende a existência de dois personagens: o autor do ataque é um minerador com acesso à criação do conteúdo do bloco sendo adicionado, a vítima, um provedor de bens ou serviços que receba criptomoedas como meio de pagamento.

Aqui, o agente malicioso gera duas transações fraudulentas: na primeira delas, ele simula realizar o pagamento por um bem em uma quantidade determinada de criptomoedas de sua transação para o provedor. Na segunda, ele desvia esta mesma quantidade de criptomoedas de uma outra conta para si.

Por ser o agente malicioso o minerador daquele bloco específico, ele consegue impedir a primeira transação — aquela que representa o pagamento original pelo bem ou serviço — de ser incluída em um bloco (pelo menos por um curto espaço de tempo, até que a segunda transação seja manipulada), e usa o dinheiro aparentemente desviado — na segunda transação — para adquirir algo do provedor de bens ou serviços. O fornecedor tem a falsa impressão de que o pagamento foi efetuado e cumpre com sua parte no acordo.

Tendo o fornecedor cumprido com sua parte no acordo, o minerador libera a primeira transação de volta para o processamento pela blockchain, que a identificará como uma transação fraudulenta e, portanto, inválida, o que invalidará também a segunda transação — fazendo com que, na realidade, o prestador acabe não recebendo seu pagamento.

8.5 Ataques Eclipse e Sybil

Os Ataques Eclipse ganharam ênfase do público após publicação de 2015 intitulada *Eclipse Attacks on Bitcoin's Peer-to-Peer Network* (Ataques Eclipse na Rede Ponto a Ponto do Bitcoin), realizada em conjunto entre participantes de Universidade de Boston e da Universidade Hebraica de Jerusalém. Eles são uma maneira relativamente simples de realização de uma ação maliciosa, agindo diretamente nos nós de uma rede.

Como seu nome sugere, um Ataque Eclipse tem como objetivo impedir que determinado nó seja capaz de ter contato com outros nós legítimos da rede. Ele é conseguido pela ligação direta de um número muito grande de nós fraudulentos a um nó, a vítima do ataque, que passa a receber informações incorretas, manipuladas, sobre a cadeia — entendo-a como a informação oficial. Assim, os agentes maliciosos conseguem enganar facilmente a vítima, podendo fraudá-la em diversos aspectos.

O Ataque *Sybil*, por sua vez, foi primeiramente identificado em publicação feita pelo time da *Microsoft*, no início dos anos 2000, assinado por John Douceur e intitulado "*The Sybil Attack*". Ele consiste de um ataque típico de redes P2P, não sendo exclusivo de ocorrer em blockchains, e considera o fato de atores maliciosos, participantes da rede distribuída, poderem personificar diferentes papéis — sem levantarem maiores suspeitas — o que,

eventualmente, termina por comprometer a capacidade de redundância destes sistemas.

Diferentemente de um Ataque Eclipse, em que o alvo é um único nó, enganado sobre o real estado da rede, um Ataque Sybil compromete a rede como um todo, podendo manipular o comportamento dos participantes ou, ao menos, influenciá-lo, e criando falsas impressões que são compartilhadas ao longo da rede, o que faz com que os outros atores sejam influenciados em sua forma de agir.

8.6 Vulnerabilidades em Smart Contracts

Vulnerabilidades em *smart contracts* são inerentes aos contratos desenvolvidos, cujo endereçamento se dá exclusivamente por meio do código e sua lógica de desenvolvimento. São encontradas com maior frequência em trechos de códigos que escapam das boas práticas recomendadas pela comunidade.

Por serem de cunho individual, isto é, por dizerem respeito ao contrato isoladamente e não ao ecossistema sobre o qual a blockchain é construída, são mais difíceis de serem controladas — já que se tornam responsabilidade do próprio desenvolvedor ou seu time — porém, podem ser mais facilmente exploradas por usuários maliciosos (possivelmente sem a necessidade de realização de ataques elaborados tecnicamente).

Em 2018, por meio do EIP-1470 o time de desenvolvimento do Ethereum criou uma Classificação de Fraquezas em *Smart Contracts* (em inglês, *Smart Contract Weakness Classification*, abreviada como SWC). Ela foi desenvolvida com as terminologias e práticas propostas pela comunidade responsável pela manutenção da lista do *Common Weakness Enumeration* (CWE, do inglês, Enumeração de Fraquezas Comuns), buscando suprir uma lacuna

existente no desenvolvimento blockchain, em que vulnerabilidades e ataques teóricos são discutidos por meio de papéis acadêmicos, artigos e guias de boas práticas, sem o envolvimento de grandes organizações comerciais.

A partir de seu repositório no GitHub, acessado no endereço https://github.com/SmartContractSecurity/SWC-registry e com livre participação, a comunidade compartilha não apenas a definição das vulnerabilidades encontradas, como lista uma série de referências e práticas de mitigação, além de apresentarem casos de teste, que podem ser usados por desenvolvedores para avaliar suas aplicações.

Dentre as vulnerabilidades listadas pelo SWC são encontradas falhas simples de código, como a SWC-100, que lida com o fato de as visibilidades não terem sido bem definidas no momento da geração de nossos contratos, até cenários complexos.

Especificamente para a SWC-100, uma vez que por padrão todo elemento do contrato é público, caso uma visibilidade não seja definida no momento da codificação, ela abre espaço para que elementos alheios ao seu contexto possam manipulá-los facilmente.

A utilização de compiladores é outro ponto básico citado na lista. Uma prática comumente encontrada em materiais públicos sobre o Solidity é o de fazer a declaração dos chamados *floating pragmas*, uma declaração *pragma* que permite a utilização de diferentes versões dos compiladores, através de declarações do tipo `>= 0.8.0` no início dos arquivos, o que indica aceitar a utilização de qualquer versão do compilador maior ou igual a 0.8.0, por exemplo. Se por um lado essa declaração permite maior flexibilidade de trabalho no código, ela também traz consigo o viés de que compiladores antigos possuem bugs conhecidos e listados e que, por poderem ser utilizados para a execução dos contratos, tornam o código vulnerável à exploração de falhas passadas.

Problemas no controle de acesso a funções — por exemplo, a chamada `selfdestruct()` — e a má realização do tratamento de exceções, por exemplo, em chamadas externas, são outras situações citadas no SWC e que lidam diretamente com a qualidade do código escrito. Situações um pouco mais complexas, como a utilização de heranças múltiplas sem um bom planejamento e conhecimento da Linearização C3, podem causar comportamentos inesperados.

Por fim, o SWC cita, ainda, falhas que possam ocorrer em condições de corridas — isto é, a dependência de dados que podem ser modificados durante o tempo de execução do código. Em uma rede blockchain, isso é especificamente importante, visto que transações levam um tempo considerável para serem incluídas em blocos e serem confirmadas; assim, dados que dependam do estado da rede, por exemplo, precisam ser bem estruturados para considerar uma mudança de estado não esperada pelo código.

A documentação do projeto Ethereum cita algumas boas práticas para serem levadas em conta durante o desenvolvimento de nossos contratos, que vão desde recomendações gerais como a utilização de versionadores de código, revisão por pares, realização de alterações via *pull requests*, até o uso de ferramentas especializadas em análise de código e a participação de auditores especializados neste tipo de tarefa. Um *linter* muito conhecido entre desenvolvedores Solidity é o *Slither*, de código aberto, que realiza uma varredura no código, identificando vulnerabilidades e explicando-as de forma intuitiva.

A ConenSys cita alguns exemplos de boas práticas e, entre elas, dá um lembrete interessante de que todos os dados processados em uma blockchain são públicos — ainda que protegidos por criptografia — fato que também deve pesar durante a escolha por um método de representação das transações. Consulte sua documentação nas referências, no capítulo final.

Outras ferramentas úteis para a visualização de estruturas, realização de *debugging* na EVM, realização de análises estáticas e dinâmicas no código, entre outros estão disponíveis em modo acessível em uma rápida pesquisa na Internet.

8.7 Ataques de reentrância

Se, por um lado, os ataques de 34% e 51% estão ligados mais à arquitetura sobre a qual a blockchain existe, ataques de reentrância são oriundos de fragilidades encontradas, principalmente, em seu código e na lógica utilizada para o processamento de suas transações.

Ataques de reentrância tornam o desenvolvimento de *smart contracts* uma tarefa extremamente sensível, sobre a qual deve ser tomada muita atenção. Em poucas palavras, ele consiste em um ataque em que um contrato malicioso é publicado e este invoca determinada função de um outro contrato — o alvo do ataque — fazendo com que o contrato malicioso seja novamente chamado, recursivamente.

A fraude ocorre uma vez que a EVM não é capaz de trabalhar com múltiplos contratos em simultaneidade, parando a execução de um contrato para iniciar a execução do outro — retornando ao contrato original após o término da nova execução. O nome reentrância advém daí: o fato de um contrato precisar ser retomado (“reentrar em execução”) constantemente após a finalização das chamadas externas.

Uma vez que o código malicioso passa a ser chamado recursivamente, o contrato original acaba não sendo executado até o final, fato este que abre a possibilidade para uma série de ações maliciosas.

A documentação do Ethereum traz um exemplo bastante claro sobre a ocorrência de um ataque de reentrância, considerando uma função que realiza a transferência de divisas e realiza três ações, nesta ordem:

1. Verifica o saldo disponível para o usuário;
2. Realiza a transferência de todo o saldo para o usuário;
3. Altera o valor de saldo disponível para zero — visto que todo o valor foi transferido previamente.

Ao realizar a transferência do saldo para o usuário, o contrato malicioso pode ser chamado, recursivamente, e, considerando que essa sua chamada foi desenhada para solicitar uma nova transferência de divisas para si, o código nunca chega até o ponto em que o valor do saldo disponível é alterado para zero. Nesse cenário, a quantia de divisas disponível é transferida para o ator malicioso sucessivamente, ainda que já não exista mais saldo real disponível na aplicação, trazendo para ele um grande benefício.

Aqui, uma mera alteração na ordem de execução do código já evitaria esse tipo de situação: caso o saldo fosse alterado para zero antes da realização da transferência, isso, por si, já não permitiria a retirada de uma quantia de dinheiro superior à disponível. Como exercício, veja como desenvolvemos o nosso método de transferência anteriormente, neste livro.

Conclusão

Este capítulo demonstrou que, apesar de a tecnologia da blockchain ser segura por natureza e, portanto, possível de ser usada em diferentes cenários que se aproveitem da sua imutabilidade, ela ainda carrega consigo uma parcela de situações de vulnerabilidade que precisam ser mitigadas. Elas envolvem não apenas a base sobre a qual um projeto é criado (no caso, a própria cadeia), mas que também sofrem interferência de fatores como a fase de desenho e codificação dos *smart contracts* e a participação dos

diferentes atores, no contexto em que nossos contratos são executados.

Situações que envolvem a segurança em blockchain são variadas, principalmente pelo fato de transações sempre envolverem a troca de divisas financeiras (e portanto, serem atrativas para personagens maliciosos). Seu manejo requer não apenas o reconhecimento de suas existências, mas também um profundo conhecimento sobre o ecossistema tecnológico utilizado pelos projetos; fatores tão específicos como a arquitetura de funcionamento da EVM, recursiva, o que normalmente passaria como um mero detalhe para a pessoa ávida em codificar, e que aqui serve como um exemplo sobre o nível de atenção que desenvolvedores devem possuir.

Com esta apresentação, dotamos você, leitora ou leitor, com uma ferramenta a mais a ser usada durante as etapas de definições de arquitetura e implementação de seus contratos, fatores estes que, seguramente, contribuirão para o sucesso de seus projetos.

Capítulo 9
Armazenamento e analytics na Rede Ethereum

9.1 Introdução

De forma proposital, deixamos para o final do livro a apresentação dos tópicos referentes ao armazenamento e consumo de arquivos no Ethereum. Isso porque, pela própria natureza da blockchain, as duas operações não são eficientes computacionalmente (além de possuírem custo financeiro elevado) e, logo, não deveriam ser o foco de aplicações desenvolvidas na rede. Este também é um capítulo mais curto que os anteriores, em parte porque são assuntos que não são tão maduros dentro do universo blockchain, e em outra por não apresentar soluções próprias do Ethereum e que, portanto, fogem ao escopo deste livro.

Grande parte da dificuldade do armazenamento na blockchain se dá devido à sua natureza distribuída, em que cada nó deve ser capaz de armazenar uma cópia inteira da cadeia. Caso ela fosse utilizada como um armazenador universal de arquivos, por exemplo, todos os nós terminariam por serem responsáveis por armazenar, em seu computador, uma cópia de todos os arquivos carregados na cadeia, de forma universal — o que tornaria isso uma tarefa impossível em um curto espaço de tempo.

Ainda assim, existem técnicas e dispositivos que são utilizados para sua realização; aplicações como o IPFS — *InterPlanetary File System* (Sistema de Arquivos Interplanetário), por exemplo, são sistemas de armazenamento comerciais, montados inteiramente de forma distribuída, e que possuem grande compatibilidade com aplicações Ethereum. E é justamente sobre estas técnicas que falaremos nas próximas seções.

Finalizando nossa apresentação, discutiremos um assunto extremamente relevante nos dias atuais: a exploração dos dados

presentes em uma blockchain. Em um contexto de *big data* tal como vivemos hoje, em que dados são gerados em grandes quantidades, o tempo todo, e por diferentes fontes, eles vêm recebendo grande atenção de desenvolvedores de tecnologias da informação, servindo para a construção de modelos matemáticos e consumidos, por exemplo, por aplicações de inteligência artificial ou sistemas especialistas que auxiliam na tomada de decisões. Sendo a blockchain um grande repositório de dados, não é de se espantar que aqui existam inovações em sua exploração e consumo, também.

9.2 Armazenamento no Ethereum

O armazenamento e o consumo de arquivos em uma blockchain são, ambos, operações extremamente ineficientes, que demandam um alto custo financeiro, além de serem prejudicadas pela própria natureza distribuída da tecnologia. Soma-se a isso um fator de segurança que não pode ser ignorado: arquivos e dados pessoais, mesmo que criptografados, estarão armazenados e acessíveis a partir de qualquer nó da rede, isto é, computadores privados, sem controle de acesso.

Vista como um elemento de armazenamento, a blockchain pode ser entendida como uma estrutura capaz de armazenar dados — e aqui, falamos de uma lista ordenada de blocos, que possuem, cada um, uma lista de transações (no Ethereum, que possui um modelo baseado em estados, cada transação representa uma mudança capaz de alterar o estado de toda a cadeia). É o chamado "Armazenamento via transações", comum a diversas blockchains.

Além do uso de transações, o Ethereum possui uma estrutura na EVM que permite a realização de armazenamento via *smart contracts*, por meio da atribuição de valor às variáveis, em um modelo chamado de *Ethereum Smart Contract Storage*

("Armazenamento em Contratos Inteligentes do Ethereum", em tradução literal). É um modelo próprio dos contratos em que cada *smart contract* possui uma implementação independente, armazenada diretamente na EVM, em um modelo lógico baseado em vetores.

1. O modelo do *Ethereum Smart Contract Storage* é baseado na existência de um vetor teórico com 2²56 posições;
2. Nele, cada posição tem capacidade para armazenamento de até 32 bytes;
3. Isso representa uma capacidade teórica de armazenamento, por contrato, de cerca de 107? bytes de dados (aproximadamente 1067 Terabytes)!

O acesso aos dados do *Ethereum Smart Contract Storage* é feito por mecanismo chave-valor. Nele, existem diferentes maneiras de armazenamento, dependendo do tipo de dado e seu tamanho (fixo ou dinâmico), ajustados em tempo de compilação ou de execução e uso de funções de *hashing*.

Variáveis de tamanho conhecido, por exemplo, são armazenadas em *slots* (espaços do vetor), na ordem que são declaradas e ajustadas em tempo de compilação. Já variáveis de tamanho desconhecido possuem diferentes formas de serem trabalhadas. Como exemplo:

1. *Arrays* dinâmicos ocupam um *slot* aleatório do *Ethereum Smart Contract Storage*, indicando seu tamanho. A seguir, os próximos slots são preenchidos em ordem, contendo os dados de cada uma das posições do vetor;

2. *Mappings* existem como *hashmaps*, que utilizam o padrão Keccak-256 como função de *hashing* (e, portanto, são operações complexas). Seu mecanismo de chave-valor via *hash* é usado para gravar e consultar dados armazenados no vetor principal.

Por ser uma operação indesejável, o armazenamento é desencorajado no Ethereum. Isso fica claro, por exemplo, quando descobrimos que operações que liberam espaço no *Ethereum Smart Contract Storage* possuem valores de *Gas* negativos associados.

Uma terceira maneira de armazenamento possibilitada pelo Ethereum é a que utiliza seus *logs* de eventos, vistos nos capítulos sobre o Solidity. Logs são, como já visto, armazenados em uma estrutura especial do bloco baseada no *Filtro de Bloom*, compostos por "tópicos" e "metadados" (máximo de 4 tópicos por evento), indexados ou não para consultas externas. Sua contrapartida é a de que não podem ser consultados e manipulados diretamente pelo *smart contract*.

9.3 Armazenamento descentralizado

Como vimos na introdução, uma grande dificuldade encontrada pela blockchain caso ela se dedicasse ao armazenamento de arquivos seria o caso da replicação de seu conteúdo entre os diferentes nós participantes da rede. Ainda que não seja o único problema a ser levantado, talvez seja o mais simples de ser citado, uma vez que, por si, acaba com as possibilidades de sua utilização caso não seja endereçado de forma adequada. Apesar de não ser uma tarefa facilmente realizada pela blockchain, existem mecanismos e técnicas que permitem sua realização sem maiores problemas.

Entre as técnicas de armazenamento em contexto de blockchain, existe uma realizada fora da cadeia (*off-chain*) — e, portanto, sem dependência direta das estruturas da EVM, por exemplo — que merece destaque: seu armazenamento é feito com base nas estruturas da tecnologia blockchain, porém de forma independente, e é classificado, portanto, como “descentralizado” (daí a expressão em inglês *dStorage — decentralized Storage*, Armazenamento descentralizado —, utilizada para se referir a ele).

No conceito do *dStorage*, temas como a utilização de mecanismos de consenso e o uso de técnicas de *hashing* são tão importantes aqui como quando falamos sobre a inclusão de transações em um bloco. No geral, as diferentes técnicas de armazenamento descentralizado se baseiam, principalmente, em alguns dos seguintes fatores:

1. Mecanismo de persistência;
2. Modo de retenção de dados;
3. Capacidade de descentralização;
4. Consenso.

O *dStorage* difere do armazenamento via contratos do Ethereum, principalmente, com respeito ao tempo de persistência dos dados — sendo o armazenamento em blockchain “eterno”, tal como qualquer transação guardada em uma cadeia de blocos (enquanto o contrato existir ou seu valor seja alterado via código), enquanto o armazenamento descentralizado é acordado por meio de pagamentos periódicos ou da participação do dono da informação na rede, terminando após o fim do acordo.

Em linhas gerais, o armazenamento de arquivos de forma descentralizada envolve uma sequência de seis passos:

1. A quebra do arquivo em pequenas partes (*shards*);
2. Aplicação de técnicas de criptografia, para garantir a segurança nos *shards*;
3. Criação de *hashes* de cada um dos *shards*;
4. Replicação dos *shards* em um número ótimo que garanta a segurança e a integridade de seu conteúdo;
5. Distribuição dos *shards* ao longo da rede;
6. Registro das operações e localizações na forma de transações em blocos ou por meios próprios.

A técnica de *sharding* é bastante comum e é utilizada mesmo em sistemas que não se baseiam em blockchain, tais como os que se dedicam ao processamento de dados, por facilitarem a sua

transferência e agilizarem seu processamento, feito em partes. Como vimos, ela também é uma técnica de grande relevância no Ethereum 2.0.

Diferentemente dos dados guardados como transações na rede, o armazenamento de arquivos como *shards* não é distribuído igualmente através de todos os nós, tendo apenas um número mínimo de segurança, que assegura que os arquivos sejam imutáveis e que persistam indefinidamente na rede.

A seguir, listamos alguns projetos comerciais que trabalham com armazenamento descentralizado, e que utilizam a estrutura da blockchain Ethereum para tal; além dos projetos citados, outros também são extremamente conhecidos e bem utilizados. Convidamos você a pesquisar sobre outros modos de armazenamento, diretamente na documentação do projeto Ethereum.

Arweave

Um importante projeto que utiliza esse tipo de armazenamento, e que é compatível com o Ethereum, é o *Arweave*. Ele possui um protocolo próprio de trabalho e cria uma rede chamada *blockweave*, cujos blocos são minerados através de um protocolo de consenso do tipo PoA, capaz de armazenar a estrutura dos dados carregados para armazenamento. Você encontra o endereço para sua documentação oficial nas referências deste livro.

IPFS

O *IPFS*, já apresentado na introdução deste capítulo, possui uma abordagem um pouco diferente, devendo o dono da informação participar da rede para garantir sua persistência. Isso é realizado através de uma ação, chamada de *pinning*.

Ao participar da rede de armazenamento e ter seus arquivos (ou seus *shards*) "fixados" (*pinned*) em sua própria máquina, o

participante garante sua permanência ali. A partir do momento em que ele é desfixado, a própria rede acaba se encarregando de eliminá-lo, quando entender que ele está ocupando espaço, o que ocorre naturalmente. No caso do *IPFS*, os endereços dos arquivos são armazenados através de um mecanismo denominado *DNSLink*.

Storj DCS

O *Storj DCS* (*Decentralized Cloud Storage*, inglês para "armazenamento em nuvem descentralizado") é um projeto que, como os anteriores, trabalha com a quebra do arquivo em pequenas partes e, da mesma forma, garante a proteção por meio de *hashes* e do uso de chaves. Porém, ele não trabalha com uma rede P2P de livre participação, mas possui uma estrutura própria, distribuída ao longo de vários países, hospedada na nuvem.

O projeto trabalha com protocolos próprios de comunicação para seleção de nós, que levam em consideração fatores como distância e períodos de latência para prover uma melhor experiência ao usuário. A rede é monetizada através do *token* STORJ, baseado na rede Ethereum.

9.4 Insights via exploradores de blocos

Exploradores de blocos são portais desenvolvidos pela comunidade de desenvolvedores Ethereum, ou por terceiros, que permitem a visualização em tempo real dos blocos sendo incluídos à rede — e, como consequência, provêm detalhes de suas transações e outros metadados.

O fato de este ser um dos últimos tópicos abordados neste livro se dá justamente porque, para bem entender a informação dada nestes portais, é necessário ter todo um conhecimento sobre o funcionamento da blockchain e, especificamente, da Rede

Ethereum, pois sua informação é baseada em dados como dificuldade, custo de *Gas*, assinaturas, *timestamps*, validadores, entre outros.

Um explorador de blocos pode ser definido de forma muito abrangente como uma ferramenta de procura desenvolvida para a blockchain, sendo capaz de apresentar os dados pertinentes sobre cada um dos blocos que compõem a cadeia. Esse tipo de tecnologia surgiu em princípios dos anos 2010, ainda para o Bitcoin, e foi se desenvolvendo ao longo do tempo.

Hoje, apesar de existirem exploradores de blocos multipropósito — isto é, capazes de darem informações sobre diferentes blockchains —, em sua maioria os principais provedores são dedicados a uma única blockchain ou, pelo menos a um nicho restrito. Assim, um explorador da Rede Ethereum, por exemplo, não necessariamente terá dados do Bitcoin, apesar de ser uma possibilidade.

Os dados incluídos em uma blockchain são todos públicos, e isso é uma das suas premissas de seu funcionamento. Como garantia de segurança, seus conteúdos são criptografados e os blocos são representados por *hashes*. Dados menos sensíveis, como endereços, porém, são públicos — logo, conhecendo determinado endereço, é possível consultar, ao mínimo, seus volumes de transação e seus principais parceiros. E isso passa a ser interessante.

Um exemplo clássico diz respeito à pessoa que comprou uma pizza e pagou com criptomoedas; a transação e seu endereço são públicas na rede. A partir do momento que o dono da pizzaria sabe para quem entregou aquele pedido, é possível determinar a localização física do dono daquele endereço, ligando ambos os dados. Aqui, vemos um exemplo extremamente simplificado de como os dados, ainda que anônimos na rede, podem ser usados até mesmo como mecanismos de segurança — como seriam em uma investigação de transações ilícitas em criptomoedas, pela polícia especializada.

Da mesma forma, os dados de transferência de divisas podem ser analisados para criar padrões e estabelecer *insights* valiosos, utilizados por cientistas de dados e profissionais de inteligência de mercado para a tomada de decisões, o que é um campo de pesquisa muito vasto, com grande potencial de ganhos financeiros, inclusive. Além disso, exploradores de bloco podem ser utilizados como monitores do mercado financeiro, para a oferta de *tokens*, entre outras coisas.

Exploradores de blocos são capazes de prover informações sobre a *mainnet*, sobre as *testnets* da blockchain, trazem informações não apenas sobre compra e venda de das criptomoedas, mas também da utilização de *tokens* (e, daí, dando indicações de preferências do usuário). Neles, é possível também ver dados ainda mais detalhados, como *uncles* (que, como vimos, são um indicador da saúde da rede, no caso do Ethereum), e ainda listam aplicações existentes, como dApps. Eles são uma forma muito confortável para lidar com aplicações Ethereum.

Em sua maioria, esses portais não acessam os dados da blockchain diretamente, já que isso traria complicações, principalmente em termos de latência. Ao invés disso, armazenam os dados do bloco de forma centralizada, em alguma espécie de banco de dados, de propriedade da própria organização, em um processo chamado “ingestão” (processo este, centralizador!).

Um importante fato é o de que a maioria dos exploradores de blocos expõe seus dados, principalmente por meio de APIs. Assim, desenvolvedores podem utilizar vários dos dados disponíveis em suas aplicações, criando inclusive mecanismos para lidar com seus contratos e a transferência de suas divisas de acordo com diferentes situações. Um portal muito conhecido de usuários da Rede Ethereum é o *Etherscan*, cujo endereço vai listado nas referências, ao final do livro.

9.5 Consultas GraphQL na blockchain

Uma das grandes inovações testemunhadas pela comunidade Ethereum nos últimos tempos foi o desenvolvimento do projeto *The Graph*, com viés de facilitar a exploração dos dados armazenados nos blocos da rede. Iniciado em 2017, ele tornou-se público em dezembro de 2020 e a partir de então passou a ser utilizado por consumidores ao redor do globo.

De acordo com a documentação do próprio projeto, o *The Graph* é definido como “um protocolo descentralizado para a indexação e consulta dos dados de uma blockchain”, sendo chamado também de “o Google da blockchain”.

Sua utilização visa diminuir as dificuldades encontradas durante as reconstituições, como aquelas realizadas diretamente por meio de exploradores de bloco, que exigem que toda a rede e toda transação em cada um dos blocos seja varrida durante a procura por dados específicos o que, apesar de ser possível, é uma tarefa muito intensa computacionalmente.

O funcionamento do *The Graph* é baseado na geração de grafos (chamados de “subgrafos”), que estabelecem relações entre os componentes de interesse, de acordo com o definido em seu manifesto, um arquivo definido em sua criação e armazenado no *IPFS*.

The Graph

O *The Graph* funciona com base em uma entidade chamada *Graph Node*, que é um nó especial na rede do projeto que utiliza o protocolo para indexar os dados da blockchain, mantendo-o atualizado através de mecanismos de *event sourcing*. Através disto, seu protocolo permite o acesso rápido aos dados de interesse.

Os dados no *The Graph* podem ser consultados através de *endpoints* expostos com capacidade de executarem consultas de GraphQL por uma interface própria chamada *GraphiQL*, em troca de *tokens* GRT, baseados no padrão ERC-20 do Ethereum.

Dentro da rede, chamada *The Graph Network*, existem quatro papéis bem definidos: o *curator*, o *indexer*, o *consumer* e o *delegator*, cada um com funções específicas dentro da rede, vistos a seguir:

1. *Curators* são nós que exploram os manifestos criados e que têm a responsabilidade de decidir quais subgrafos são necessários existirem dentro da rede. São recompensados, proporcionalmente, com o sucesso do consumo de determinado subgrafo e, portanto, visam sempre fazerem as melhores escolhas como forma de maximizar seus lucros. Como nem todos dados podem despertar o interesse por parte dos curators, os próprios desenvolvedores podem assumir esse papel, através do pagamento em *tokens* GRT.

2. *Indexers* são os nós operativos dentro da *The Graph Network* e são recompensados pelos seus serviços de indexação. São nós que se dedicam a olhar para a *mainnet* e criar os subgrafos, de acordo com os dados de interesse estabelecidos pelos *curators*.

3. *Consumers* são usuários que desejam usar os serviços de consulta e que pagam divisas financeiras para os *indexers* funcionarem. Esses serviços são negociados em *marketplaces* especializados.

4. *Delegators*, por sua vez, são participantes que não desejam agir como nós do *The Graph*, porém, que desejam contribuir, de alguma forma, com a rede. Através de sua participação eles aumentam a segurança da rede, através do fornecimento de divisas para os *indexers*, recebendo, em troca, parte das divisas recebidas como recompensa pelos serviços de indexação.

Ao realizar a indexação de um subgrafo, o *indexer* deve garantir seu trabalho através de um mecanismo de consenso chamado *Proof of Indexing* (PoI), uma forma de assinatura eletrônica que garante seu trabalho, e que é exclusiva daquele subgrafo — tal como um mecanismo de PoW funciona para determinado bloco. Disputas, quando levantadas, envolvem a presença de figuras com ação de mediação, que definem se determinado trabalho de indexação foi feito ou não corretamente através de mecanismos *trustless* e descentralizados, tal como a própria ideologia da blockchain.

O projeto *The Graph* permite a realização de consultas diretamente através de seu site, em uma área denominada *playground*, dentro da sua solução *Graph Explorer*, familiar para desenvolvedores que alguma vez já trabalharam com *GraphQL* e quiseram realizar testes, e também através de APIs desenvolvidas em seu *Graph Studio*, estas podendo ser acessadas diretamente via código, inclusive pela *stack* web3 apresentada neste livro.

A exploração de dados em uma blockchain é, sem dúvidas, uma área que se desenvolverá muito nos próximos anos, o que traz oportunidades bastante grandes dentro deste mercado. A *The Graph Network* é uma tecnologia que vem se destacando neste ínterim, revolucionando a forma com que os dados são tratados dentro do contexto descentralizado sendo, portanto, ferramenta fundamental a ser conhecida por pessoas que desejem utilizar aplicações no Ethereum.

9.6 Conclusão

Com a finalização deste capítulo, concluímos, também, a apresentação dos tópicos propostos a serem abordados neste livro. Sendo esta uma publicação sobre fundamentos, planejamos cuidadosamente expor cada um dos assuntos pertinentes, fazendo menções a temas relevantes não apenas ao Ethereum — nossa

principal rede de interesse — mas também ao universo de tecnologias com base na blockchain, em forma gradual e dividida de acordo com os contextos de interesse.

Neste ponto, esperamos que você se sinta verdadeiramente confortável para discutir a blockchain, conseguindo argumentar seus usos e possibilidades com base verdadeiramente técnica — e aqui, reforçamos a ideia de que o livro tem um público-alvo bastante amplo.

Ao longo dos últimos capítulos exploramos com detalhes temas como o funcionamento de sistemas descentralizados no geral, e do Ethereum, como nossa blockchain de trabalho. Vimos suas aplicações e possibilidades, limitações, detalhes de sua arquitetura e de seus componentes, apresentamos desafios enfrentados por quem opta pelo seu uso, passamos por uma parte dedicada exclusivamente à codificação, com a apresentação de uma nova linguagem de programação — o *Solidity* — com exemplos de uso em implementações de *smart contracts* e *tokens*.

Por fim, discutimos temas importantes de segurança e vulnerabilidades em trechos de código e finalizamos nossa apresentação discorrendo sobre alguns dos principais projetos que trabalham com o armazenamento e exploração de dados no Ethereum, tecnologias novas, com muito potencial de crescimento no futuro próximo. Para uma jornada sobre fundamentos, estamos satisfeitos com a quantidade de conhecimento compartilhado!

Como tudo no mundo da TI, nossa blockchain evolui muito rápido. Temos uma comunidade muito ativa, e projetos que têm sido extremamente bem-sucedidos, que trazem soluções e novos desafios a todo momento. Para você, que agora está preparada(o) para sair da etapa de fundamentos e começar a se especializar em *frameworks* e soluções próprias para cada contexto, fica, primeiramente, nosso agradecimento pela confiança e, em segundo lugar, o convite a revisitar este livro sempre que sentir necessidade. Certamente nosso livro, que se dispôs desde o primeiro momento a

apresentar as bases do Ethereum, será útil para ajudar no planejamento e na revisão de estratégias do uso da blockchain para seus projetos reais, em ambientes de produção.

Capítulo 10
Referências

A seguir listamos algumas referências citadas durante a escrita do livro, e também um material suplementar, que segue como sugestão de leitura. Todos os sites foram visitados entre julho e setembro/2021.

101 BLOCKCHAINS. *History of Blockchain Technology:* A Detailed Guide (por Gwyneth Iredale, 2020). Disponível em https://101blockchains.com/history-of-blockchain-timeline/

AMAZON WEBSERVICES. Blockchain na AWS (documentação oficial). Disponível em https://aws.amazon.com/pt/blockchain/

ANTONOPOULOS, Andreas M.; MOOD, Gavin. *Mastering Ethereum* (2018). O'Reilly Media, inc. ISBN 9781491971949.

ARWEAVE. Technology (documentação oficial). Disponível em https://www.arweave.org/technology

ASGAONKAR, Aditya. *Casper CBC, Simplified!*. Disponível em https://medium.com/@aditya.asgaonkar/casper-cbc-simplified-2370922f9aa6

ASGAONKAR, Aditya. *Casper FFG Explainer* (2020). Disponível em https://www.adiasg.me/2020/03/31/casper-ffg-explainer.html

BEINCRYPTO. Brasileira lança mercado NFT de música e primeira faixa arrecada R$ 60 mil em 24 horas (por Saori Honorato). Disponível em https://beincrypto.com.br/brasileira-lanca-mercado-nft-de-musica-e-primeira-faixa-arrecada-r-60-mil-em-24-horas/

BLOCKTELEGRAPH. *Blockchain Before Bitcoin:* A History (por Stefen Beyer). Disponível em https://blocktelegraph.io/blockchain-before-bitcoin-history/

BRASIL. Lei Nº 13.709, de 14 de agosto de 2018 — Lei Geral de Proteção de Dados Pessoais (2018). Disponível em http://www.planalto.gov.br/ccivil_03/_ato2015-2018/2018/Lei/L13709.htm#art65

BRASIL. Ministério da Defesa: Proteção de Dados — LGPD (2020). Disponível em https://www.gov.br/defesa/pt-br/acesso-a-informacao/lei-geral-de-protecao-de-dados-pessoais-lgpd

CHIU, Juin. *Casper FFG:* Consensus Protocol for the Realization of Proof-of-Stake (2018). Disponível em https://medium.com/unitychain/intro-to-casper-ffg-9ed944d98b2d

CNN BUSINESS. *What is an NFT?* Non-fungible tokens explained (por Jazmin Goodwin, 2021). Disponível em https://edition.cnn.com/2021/03/17/business/what-is-nft-meaning-fe-series/index.html

COINCENTRAL. *A Comprehensive Guide to the Different Ethereum Token Standards* (por Ben Whittle, 2019). Disponível em https://coincentral.com/a-comprehensive-guide-to-the-different-ethereum-token-standards/

CONSENSYS. *Ethereum Smart Contract Best Practices* (documentação oficial). Disponível em https://consensys.github.io/smart-contract-best-practices/recommendations/

CONSENSYS. *Ethereum Smart Contract Security Best Practices* (documentação oficial). Disponível em https://consensys.github.io/smart-contract-best-practices/

CONSENSYS. *Getting Started with Quorum Blockchain Service from ConsenSys* (documentação oficial). Disponível em https://consensys.net/QBS

CRYPTO BULLS CLUB. *The DAO Attack:* What is it and What Went Wrong? (por Srujana M N, 2021). Disponível em

https://cryptobullsclub.com/the-dao-attack/

CRYPTO NEWS AUSTRALIA. *DeFi Total Value Locked Has Exploded This Year, Up 10x Since January* (por José Oramas, 2021). Disponível em https://cryptonews.com.au/defi-total-value-locked-has-exploded-this-year-up-10x-since

CRYPTONEWS. *Clearing Up Casper, Proof of Stake and Beacon Chain Confusion, Once And for All*. Disponível em https://cryptonews.net/en/news/ethereum/83545/

CRYPTOPEDIA. *Decentraland:* A Virtual World Built on Ethereum (2021). Disponível em https://www.gemini.com/cryptopedia/decentraland-defi-ethereum-based

CRYPTOPEDIA. *What Was The DAO?* (2021). Disponível em https://www.gemini.com/cryptopedia/the-dao-hack-makerdao

CRYPTOTICKER. *What are Ethereum Tokens?* (por Lorenzo Stroe, 2019). Disponível em https://cryptoticker.io/en/what-ethereum-tokens/

DECRYPT. *Dapp volume hits $12 billion as Ethereum dominates* (por Greg Thomson, 2021). Disponível em https://decrypt.co/34494/dapp-volume-hits-12-billion-as-ethereum-dominates

DELLOITE. *Delloite's 2020 Global Blockchain Survey* (2021). Disponível em https://www2.deloitte.com/content/dam/insights/us/articles/6608_2020-global-blockchain-survey/DI_CIR%202020%20global%20blockchain%20survey.pdf

ETH GAS STATION (documentação oficial). Disponível em https://ethgasstation.info/

ETHEREUM (documentação oficial). Disponível em https://ethereum.org/

ETHEREUM. Solc JS (documentação oficial). Disponível em https://github.com/ethereum/solc-js

ETHEREUM. Solidity Compiler (documentação oficial). Disponível em https://github.com/ethereum/solc-bin

ETHEREUM WIKI. Ethash. Disponível em https://eth.wiki/en/concepts/ethash/ethash

ETHERSCAN (documentação oficial). Disponível em https://etherscan.io/

ETHHUB. Decentralized Finance (documentação oficial). Disponível em https://docs.ethhub.io/built-on-ethereum/open-finance/what-is-open-finance/

ETHHUB. What is Ether? (documentação oficial). Disponível em https://docs.ethhub.io/ethereum-basics/what-is-ether/

EXAME. Britânico que perdeu HD com R$ 1 bilhão em bitcoin tem novo plano de busca (por Gabriel Rubinstein, 2021). Disponível em https://exame.com/future-of-money/criptoativos/britanico-que-perdeu-hd-com-r-1-bilhao-em-bitcoin-tem-novo-plano-de-busca/

FINEMATICS. *DEFI* - The Future of Finance Explained. Disponível em https://www.youtube.com/watch?v=H-O3r2YMWJ4

HABER, Stuart; STORNETTA, W. Scott. *How to Time-Stamp a Digital Document* (1991). Disponível em https://www.anf.es/pdf/Haber_Stornetta.pdf

HARVARD BUSINESS REVIEW. *A Brief History of Blockchain* (por Vinay Gupta, 2017). Disponível em https://hbr.org/2017/02/a-brief-history-of-blockchain

HEIJE, Wesley Van. *Clean Contracts* — A guide on smart contract patterns & practices (2020). Disponível em https://www.wslyvh.com/clean-contracts/

HYPERLEDGER. *An Introduction to Hyperledger* (2018). Disponível em https://www.hyperledger.org/wp-content/uploads/2018/08/HL_Whitepaper_IntroductiontoHyperledger.pdf

LARVA LABS. *CryptoPunks*. Disponível em https://www.larvalabs.com/cryptopunks

LO, Sing Kuang; XU, Xiwei; CHIAM, Yin Kia e LU, Qinghua. *Evaluating Suitability of Applying Blockchain* (2017). *22nd International Conference on Engineering of Complex Computer Systems (ICECCS)*. Disponível em https://ieeexplore.ieee.org/document/8292816

MICROSOFT. *The Byzantine Generals Problem* (por Leslie Lamport, Robert Shostak e Marshall Pease, 1982). Disponível em https://www.microsoft.com/en-us/research/uploads/prod/2016/12/The-Byzantine-Generals-Problem.pdf

MICROSOFT. *The Sybil Attack* (por John R Douceur, 2002). Disponível em https://www.microsoft.com/en-us/research/wp-content/uploads/2002/01/IPTPS2002.pdf

MOCHA (documentação oficial). Disponível em https://mochajs.org/

MOREIRA, Mário Aurélio Ribeiro. *ECDSA (Elliptic Curve Digital Signature Algorithm)* (2006). Disponível em http://www.inf.ufsc.br/~bosco.sobral/ensino/ine5630/material-cripto-seg/crpt_trabalho_ecdsa.pdf

MY CRYPTO. *The Magic of Digital Signatures on Ethereum* (por Maarten Zuidhoorn, 2020). Disponível em https://medium.com/mycrypto/the-magic-of-digital-signatures-on-ethereum-98fe184dc9c7

OLHAR DIGITAL. *Phonogram.me:* conheça a primeira plataforma de NFT do Brasil focada em música (por Tissiane Vincentin, 2021).

Disponível em https://olhardigital.com.br/2021/03/22/pro/phonogram-me-conheca-a-primeira-plataforma-de-nft-do-brasil-focada-em-musica/

PLASMA. *Plasma:* Scalable Autonomous Smart Contracts (por Joseph Poon e Vitalik Buterin, 2017). Disponível em https://plasma.io/plasma.pdf

POLKADOT WIKI. *Polkadot and Cosmos* (documentação oficial). Disponível em https://wiki.polkadot.network/docs/learn-comparisons-cosmos

KASIREDDY, Preethi. *How does Ethereum work, anyway?* (2017). Disponível em https://preethikasireddy.medium.com/how-does-ethereum-work-anyway-22d1df506369

SATOSHI NAKAMOTO INSTITUTE. *Bitcoin:* A Peer-to-Peer Electronic Cash System (por Satoshi Nakamoto, 2008). Disponível em https://nakamotoinstitute.org/bitcoin/

SATOSHI NAKAMOTO INSTITUTE. *Formalizing and Securing Relationships on Public Networks* (por Nick Szabo, 1997). Disponível em https://nakamotoinstitute.org/formalizing-securing-relationships/

SOCIEDADE BRASILEIRA DE MATEMÁTICA. *Curvas Elípticas sobre Corpos Finitos e Criptografia de Chave Pública* (2009). Disponível em https://www.sbm.org.br/docs/coloquios/CO-1-04.pdf

SOLIDITY (documentação oficial). Disponível em https://docs.soliditylang.org/

STORJ. Introducing Storj DCS (documentação oficial). Disponível em https://storj.io/how-it-works

SWC. SWC Registry (documentação oficial). Disponível em https://swcregistry.io/

TECHFUNNEL. *Blockchain Storage:* Meet Your Storage Needs. Disponível em https://www.techfunnel.com/information-technology/blockchain-storage/

THE COINBASE BLOG. *A Beginner's Guide to Decentralized Finance (DeFi)*. Disponível em https://blog.coinbase.com/a-beginners-guide-to-decentralized-finance-defi-574c68ff43c4

THE LINUX FOUNDATION. SPDX License List (documentação oficial). Disponível em https://spdx.org/licenses/

TRUFFLE SUITE. Ganache Overview (documentação oficial). Disponível em https://www.trufflesuite.com/docs/ganache/overview

USENIX SECURITY SYMPOSIUM. *Eclipse Attacks on Bitcoin's Peer-to-Peer Network* (por Ethan Heilman, Alison Kendler, Aviv Zohar e Sharon Goldberg, 2015). Disponível em https://eprint.iacr.org/2015/263.pdf

VOLLAND, Franz. *Solidity Patterns* (2017). Disponível em https://fravoll.github.io/solidity-patterns/

ZAGO, Matteo. *Why the Web 3.0 Matters and you should know about it* (2018). Disponível em https://medium.com/@essentia1/why-the-web-3-0-matters-and-you-should-know-about-it-a5851d63c949

ZEPHYRNET. *SushiSwap DEX Integra-se ao Avalanche*. Disponível em https://zephyrnet.com/pt/ethereums-lideram-c%C3%A2mbio-descentralizado-rivais-avalanche/

ZERO EXCHANGE. *Introducing the Polygon* (MATIC) bridge (2020). Disponível em https://medium.com/@OfficialZeroDex/introducing-the-polygon-matic-bridge-145f7909fed1